



**Calhoun: The NPS Institutional Archive**  
**DSpace Repository**

---

Theses and Dissertations

1. Thesis and Dissertation Collection, all items

---

1990

# The identification of software failure regions

Bolchoz, John Manning

Monterey, California: Naval Postgraduate School

---

<http://hdl.handle.net/10945/27720>

---

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

*Downloaded from NPS Archive: Calhoun*



Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

**Dudley Knox Library / Naval Postgraduate School**  
**411 Dyer Road / 1 University Circle**  
**Monterey, California USA 93943**

<http://www.nps.edu/library>

AD-A223 058

FILE COPY

2

# NAVAL POSTGRADUATE SCHOOL

## Monterey, California



DTIC  
JUN 22 1990  
nu

## THESIS

THE IDENTIFICATION  
OF SOFTWARE FAILURE REGIONS

by

John Manning Bolchoz

June, 1990

Thesis Advisor:

Timothy Shimeall

Approved for public release; distribution is unlimited.

90 06 22 120

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

## REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION <b>UNCLASSIFIED</b>		1b. RESTRICTIVE MARKINGS	
2a. SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited	
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE		5. MONITORING ORGANIZATION REPORT NUMBER(S)	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)		7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School	
6a. NAME OF PERFORMING ORGANIZATION Computer Technology Curriculum Naval Postgraduate School	6b. OFFICE SYMBOL (if applicable) 37	7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943	
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943		9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8a. NAME OF FUNDING/SPONSORING ORGANIZATION	8b. OFFICE SYMBOL (if applicable)	10. SOURCE OF FUNDING NUMBERS	
8c. ADDRESS (City, State, and ZIP Code)		PROGRAM ELEMENT NO.	PROJECT NO.
		TASK NO.	WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) <b>THE IDENTIFICATION OF SOFTWARE FAILURE REGIONS(U)</b>			
12. PERSONAL AUTHOR(S) <b>BOLCHOZ, John Manning</b>			
13a. TYPE OF REPORT Master's Thesis	13b. TIME COVERED FROM TO	14. DATE OF REPORT (Year, Month, Day) June 1990	15. PAGE COUNT 99
16. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
17. COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)	
FIELD	GROUP	SUB-GROUP	
		Software Testing, Code Analysis, Data Selection, Software Failure Region, Reachability, Error Generation, Error Propagation, Masking	
19. ABSTRACT (Continue on reverse if necessary and identify by block number) In these days of spiralling software costs and the proliferation of computers, software testing during development is now recognized as a critical aspect of the software engineering process, an aspect that must be improved in terms of cost and timeliness. This thesis describes one method that may guide software testing by analyzing the regions of input associated with each fault as it is detected. These failure regions are defined and a method of failure region analysis is described in detail. The thesis describes how this analysis may be used to detect non-obviously redundant test cases. A preliminary examination of the manual analysis method is performed with a set of programs from a prior reliability experiment. Based on the faults discovered during the previous experiment, this thesis defines the reachability conditions, the error generation conditions, and the conditions in which an error is not masked by later processing. The manual analysis of failure regions can be a difficult process, with difficulty dependent on program size, program complexity, and the size of the input data space. Program constructs and events that simplify the			
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS		21. ABSTRACT SECURITY CLASSIFICATION <b>UNCLASSIFIED</b>	
22a. NAME OF RESPONSIBLE INDIVIDUAL Timothy J. Shimeall		22b. TELEPHONE (Include Area Code) (408) 646-2509	22c. OFFICE SYMBOL CS/Sm (52Sm)

**19 Abstract (Continued)**

analysis process are also described. The thesis explains variable contamination and the effects of vertical and horizontal contamination. The thesis also describes the indirect benefits of performing failure region analysis. Finally, there are several open questions raised by this research, and these questions are presented as ideas for future research.

Approved for public release; distribution is unlimited.

The Identification of Software Failure Regions

by

John Manning Bolchoz  
Major, United States Army  
B.S., United States Military Academy, 1978

Submitted in partial fulfillment  
of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

June 1990

Author:

John Manning Bolchoz

Approved by:

Timothy Shimeall, Thesis Advisor

LCDR Rachel Griffin, Second Reader

Robert B. McGhee, Chairman  
Department of Computer Science




Accession For	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
and/or	
Initial	
A-1	

## ABSTRACT

✓ In these days of spiralling software costs and the proliferation of computers, software testing during development is now recognized as a critical aspect of the software engineering process, an aspect that must be improved in terms of cost and timeliness. This thesis describes one method that may guide software testing by analyzing the regions of input associated with each fault as it is detected. These software failure regions are defined and a method of failure region analysis is described in detail. The thesis describes how this analysis may be used to detect non-obviously redundant test cases.

A preliminary examination of the manual analysis method is performed with a set of programs from a prior reliability experiment. Based on faults discovered during the previous experiment, this thesis defines the *reachability conditions*, the *error generation conditions*, and the conditions in which an error is not masked by later processing.

The manual analysis of failure regions can be a difficult process, with difficulty dependent on program size, program complexity, and the size of the input data space. Program constructs and events that simplify the analysis process are also described. The thesis explains variable contamination and the effects of vertical and horizontal contamination. The thesis also describes the indirect benefits of performing failure region analysis. Finally, there are several open questions raised by this research, and these questions are presented as ideas for future research.



## TABLE OF CONTENTS

I. BACKGROUND AND RELATED RESEARCH .....	1
A. INTRODUCTION .....	1
B. FAILURE REGIONS .....	4
C. SOFTWARE TESTING .....	6
1. General .....	6
2. The Software Development Cycle .....	7
3. Testing Methods .....	9
a. Test Methods Related to Failure Analysis .....	9
(1) Walkthrough Testing .....	10
(2) Mutation Testing .....	11
(3) Module Testing .....	12
(4) Functional and Structural Testing .....	13
(5) Regression Testing .....	14
(6) Fault Sensitivity Analysis .....	15
b. Benefits of failure region application .....	16
D. OVERVIEW .....	16

II. METHODOLOGY FOR ANALYZING THE FAILURE REGION . . . . .	18
A. SPECIFICATION AND PROGRAM SOURCE CODE DESCRIPTION . . . . .	18
B. ASSUMPTIONS AND PRECONDITIONS . . . . .	19
C. MANUAL METHODOLOGY FOR ANALYZING FAILURE REGIONS . . . . .	22
1. General . . . . .	22
2. Development of the Reachability Conditions (Condition I) . . . . .	23
3. Development of the Error Generation Conditions (Condition II) . . . . .	30
4. Development of the Conditions Under Which the Fault Is Not Masked (Condition III) . . . . .	32
5. Example of Defining the Entire Failure Region . . . . .	34
D. CONCLUSION . . . . .	37
III. OBSERVATIONS AND FINDINGS . . . . .	39
A. INTRODUCTION . . . . .	39
B. OBSERVATIONS . . . . .	40
1. FAILURE REGION OBSERVATIONS . . . . .	40
2. PROCESS OBSERVATIONS . . . . .	42
a. General . . . . .	42
b. Comparative Difficulty of Determining Failure Region Conditions . . . . .	42



c.	Common Causes for Generating TRUE Conditions . . . . .	45
(1)	Error Reachability Conditions . . . . .	46
(2)	Error Generation Conditions . . . . .	46
(3)	Conditions Under Which a Bad Value Isn't Masked by Later Processing . . . . .	47
d.	Common Ground Between Failure Region Conditions . . . .	47
e.	Variable Contamination . . . . .	49
IV.	CONCLUSIONS AND RECOMMENDATIONS FOR FUTURE RESEARCH . . . . .	50
A.	CONCLUSIONS . . . . .	51
B.	RECOMMENDATIONS FOR FUTURE RESEARCH . . . . .	52
APPENDIX A	. . . . .	55
APPENDIX B	. . . . .	58
APPENDIX C	. . . . .	64
APPENDIX D	. . . . .	66
APPENDIX E	. . . . .	68

APPENDIX F .....	71
LIST OF REFERENCES .....	82
BIBLIOGRAPHY .....	84
INITIAL DISTRIBUTION LIST .....	87

## LIST OF FIGURES

Figure 2.1	Range Checks on Initialized Variables . . . . .	25
Figure 2.1	Requirements Defining the Reachability Conditions . . . . .	26
Figure 2.3	External Reachability Conditions . . . . .	28
Figure 2.4	Reachability Conditions (Condition I) . . . . .	29
Figure 2.5	Source Code Example . . . . .	32
Figure 2.6	Internal Reachability Conditions . . . . .	35
Figure 2.7	The Complete Failure Region . . . . .	38
Figure 3.1	Example of Duplication of Reachability Conditions . . . . .	44

## ACKNOWLEDGEMENTS

I thank Timothy Shimeall for all of his support, encouragement, and coaching throughout the last two years both as an instructor and as a thesis advisor. His guidance and assistance were invaluable in bringing my research to its successful conclusion. Thanks also to Rachel Griffin for her very helpful comments and suggestions.

I would also like to thank my wife, Maureen, and my sons, Manning, Ryan, and Patrick, for their patience and support during our two years at the Naval Postgraduate School.

## **I. BACKGROUND AND RELATED RESEARCH**

### **A. INTRODUCTION**

In this technologically advanced age of space and underwater exploration, rapid mass transit, reliable global communications, and expanding research and development into the unknown, computers are having an increasing impact on our every day lives. The role of computers have come to permeate every aspect of human life - the cars that we drive to work, the elevators that we take to our offices, the appliances that we use to cook our food, the entertainment that we enjoy, and the various means that we use to communicate with each other are all heavily influenced by computers.

When IBM unbundled their software in 1969 by producing and selling their hardware and software separately, this contributed to the growth of a software industry which is now flourishing (Shelly and Cashman, 1984, p.17.2). As early as 1976, the expense incurred in producing and maintaining software exceeded ten billion dollars and the joint revenues of software suppliers exceeded one billion dollars (Bahr, 1980, p.1). In a Department of Defense planning document, it was discovered that 80% of current and future DoD programs in the 1985 - 1989 time frame would contain a significant software component, and by 1990, 85% of DoD's embedded systems would be allocated to software (Cavano, 1985, p.1449). In 1984, the computer industry was comprised of more than 10,000 computer companies with revenues in excess of 75 billion dollars. Shelly and Cashman predicted that between 1984 and 1989 the software industry would grow

at a rate exceeding 25% a year and that software sales would exceed 30 billion dollars (Shelly and Cashman, 1984, p.17.4).

As computers are used increasingly in critical commercial, on-line, and real-time applications, the demand for reliable, fault tolerant software systems becomes more critical. The issues of software testing and fault tolerance are becoming increasingly serious as software systems become larger and more complex, as computer systems become independent of human input, and as safety becomes more software dependent. Blum argues that "developing better computing systems, e.g., safer, more reliable, more secure, is an issue that software engineers must address (if not solve)" (Blum, 1989, p.1).

Computer involvement in our daily lives has reached the point where its impact goes virtually unnoticed until something goes wrong. It is important to point out that safety in computer dependent systems is threatened by common, seemingly simple faults as well as more serious faults. For the purposes of this paper, a fault is defined as an accidental condition that causes a functional unit to fail to perform its required function. An error is a discrepancy between a computed, observed or measured value or condition and the true, specified or theoretically correct value or condition. A failure is the termination of the ability of a functional unit to perform its required function. (Glossary, 1983) The presence of faults in program code determines the failures that software can experience. In their study of fault sensitivity, Voas and Morell show that there is no simple relationship that describes the impact a software fault can have on the failures in a program. (Voas and Morell, Dec 1989, p.1) In its cumulative list of computer failures, the Special Interest Group on Software Engineering (SIGSOFT) details over 500 computer

failures in 22 areas of every day life. An examination of this list also reveals that many of these tragic and expensive failures were caused by 'simple mistakes.' (Neumann, 1989, RISKS, pp. 5 - 21)

Based on the premise that common faults can be as critical as serious faults, an efficient method for finding common faults is imperative. One approach to this problem is to study the occurrence of errors in proximity to other errors. Known as error clustering, Myers describes this phenomenon as the probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that section (Myers, 1979, p.15). In order to develop this fault finding method, it is necessary to understand how test data causes these faults to produce software failures. In other words, it is necessary to determine what regions of the program's input space are mapped by faults in the program source code to failures in the output space. These regions have been called 'failure regions.' (Ammann and Knight, 1988, p. 419)

While there have been several studies which addressed the concept of failure regions, none of these studies analyzed failure regions in light of improving testing efficiency, and none have discussed the derivation of failure regions from faults identified in the program source code. This thesis is an analysis of the process of identifying failure regions, evaluating how this process can be made more efficient.

The ultimate goal of this research is an in-depth understanding of failure regions and their impact on the testing process. Additionally, analysis techniques will be used to examine the conditions bounding the failure regions. Finally, this research will also serve as a foundation for further research efforts in the areas of error clustering and

failure regions. Prior to examining the specifics of failure regions, a brief summary is presented of relevant previous work in the field of software testing.

## **B. FAILURE REGIONS**

As stated, there has been a significant void in the area of failure regions and error clustering, however, the idea of manipulating the input data to refine the testing process is clearly gaining momentum. Shimeall and Griffin define a software failure region as

that portion of a program input space that is mapped by a program defect into a failure or erroneous program result. This region is bounded by inequalities deriving the combination of three sources: the reachability conditions for the code with the defect, the conditions under which the calculations in that code produces an erroneous value, and the conditions in which the value isn't masked by later processing. (Shimeall and Griffin, 1989, p.1)

It is important to note that the failure region is part of the input space and not the fault itself. The failure region is eventually mapped into a program failure. Once the program fault is identified, the program source code can be analyzed to determine the failure region. Once the failure region is isolated, any test cases that fall within that region can be omitted from the testing process until the fault is corrected, while other test cases can proceed. Once the fault is removed by modification of the program source code, the failure region can be used as a guide for testing the correctness of the modification. This process allows for the elimination of redundant test cases and permits detection of multiple faults. Subsequently, this process allows for improvement of the initial testing process as well as any subsequent regression testing that is conducted (Shimeall and Griffin, 1989, pp. 1,2).



The concept of the application of failure regions to the software engineering process was first introduced by Ammann and Knight in their work on data diversity and fault tolerance. Ammann and Knight's work centered on the use of data redundancy as a means of ensuring fault tolerance, a fault tolerant strategy that complements design diversity. The failure domain of the program i.e., the set of input data that causes the program to fail and its geometry, comprise what Ammann and Knight define as the failure region. (Ammann and Knight, 1988, pp.418,419)

The success of the data diversity depends on the ability of a reexpression algorithm to produce data points outside of the failure region, given an initial set of data points inside of the failure region. The reexpression algorithm transforms an original set of input data into a new but equivalent set of input data. An input  $x$  is provided to a program  $P$  that produces the output  $P(x)$ . A reexpression algorithm then transforms the original input  $x$  to produce a new input  $y$  where  $y = R(x)$ . Either concurrently or otherwise, the program  $P$  operates on both  $x$  and  $y$  to produce the output  $P(x)$  and  $P(y)$ . The reexpression algorithm is considered valid as long as the original information content is preserved. (Ammann and Knight, 1988, p.419)

Although the initial study on data diversity produced a wide margin of performance, and empirical results have not conclusively proven the effectiveness of this method, Ammann and Knight's initial results did show some success and support further research in the area. Additionally, the proven success of design diversity lends further confidence to the idea that since diversity in the design space may provide fault tolerance, diversity in the data space may do the same (Ammann and Knight, 1988, p.418).

While this research will not involve the specific issue of fault tolerance that Anmann and Knight explored, manipulation of the input data and examination of failure regions in the testing process will be the critical aspect of this thesis research.

## **C. SOFTWARE TESTING**

### **1. General**

There has been an extensive amount of research devoted to software testing, and the significance and importance of testing in the overall software development process is being recognized as more and more critical. Once considered only a necessary function testing is now being viewed as more important in reducing future maintenance costs.

In 1976, Alberts estimated that up to 50% of development cost is incurred during the testing phase, and as much as 90% of a product's total life-cycle costs involve maintenance to correct errors and revise the software to meet new requirements (Alberts, 1976). In 1979, Myers estimated that in a typical programming project, approximately 50% of the time and more than 50% of the total costs are devoted to the testing phase (Myers, 1979, p.vii). Cavano, in his work on high confidence software for the DoD in 1985, suggested that a full 40% of the overall development effort should be devoted to testing (Cavano, 1985, p.1454).

While the exact figures are not critical, it should be clear that testing is a vital component of the software development cycle and the product life-cycle. As such, it is imperative that research on software testing be "put on the front burner" as the critical

issue facing software development in the 1990's. In this regard, several DoD sponsored study teams conducted evaluations to determine how the military could improve its management of computer resources. One of their conclusions was a failure on the part of the military "to prescribe and adhere to a disciplined hardware and software development methodology" (Reifer, 1977, p.125). As a result of this study, the DoD issued a directive to all service components to "develop and implement a (sic) approach disciplined to the management of software design, engineering and programming which will ensure the provision of effective software at minimum life-cycle cost" (Reifer, 1977, p.125).

## **2. The Software Development Cycle**

One method of determining program correctness and one that propels much of the software engineering research effort is the concept of program testing. Testing has been defined as the process of executing a program with the specific intent of finding errors or faults (Myers, 1979, p.16). A fault is defined as "an accidental condition that causes a functional unit to fail to perform its required function" (Glossary, 1983). A good test case has been described as one that has a high probability of detecting errors (Myers, 1979, p.16).

The problem is that it is virtually impossible at this point in the evolution of testing to determine what successful testing is and when it has been achieved. The difficulty is that in order to be fully confident that testing is complete and successful, it is necessary to test exhaustively all possible executions of the program with all possible data inputs. This is difficult in a small program and economically and realistically

infeasible, if not impossible, for larger programs. It is this inability to test exhaustively that drives the research in software testing - the analysis of existing testing methods and the search for better methods.

It is almost universally agreed upon that testing must be viewed as the destructive process of finding errors and this must be the approach taken into the testing phase. The testing agency/individual must aggressively approach the testing task with the goal of discovering errors, not with the goal of showing that the program works correctly.

Bahr points out that prior to the establishment of formal testing methods, programmers established test data sets in hopes of discovering errors, thereby ensuring proper program execution for that data considered representative of real system use. This allowed for the fielding of software systems that contained many undiscovered performance errors. This also caused many organizations to formalize more aggressive, multi-dimensional test strategies to test the functional and performance requirements of the system under test prior to release. (Bahr, 1980, p.21) Many studies comparing test strategies continue to support the idea that a thorough test plan must encompass multiple test strategies and not focus on one particular method (Myers, 1979; Bahr, 1980; Shimeall and Leveson, 1989).

Many software engineers also recommend that all testing be conducted by a person or persons other than those involved with the design and/or implementation of the code. It must be an unbiased effort which should not involve egos or personal feelings. Furthermore, testing must be conducted not only to determine if a program does not do

what it is supposed to do, but the program must also be tested to ensure that it does what it is not supposed to do (Myers, 1979, p.7).

The study of failure regions and their application to the testing process is not without precedent or rationale. The concept of boundary value testing provides anecdotal evidence that failure regions do not occur randomly and that the evaluation of these input data sets is critical to the testing process. Boundary value analysis is the evaluation of test cases that explore the boundary conditions of the program - those conditions directly on, above, and beneath the edges of the input equivalence and output equivalence classes. Boundary value analysis is a popular test strategy, applied to both small and large projects, that relies on selection of specific input data and provides for rapid identification of data sets or regions that cause a program to execute either correctly or incorrectly.

### **3. Testing Methods**

#### ***a. Test Methods Related to Failure Analysis***

In his text on software testing techniques, Beizer points out that program testing comprises half of the development labor required to produce a working program (Beizer, 1983, p.4). It has become clear, however, that a formal, well-defined test plan is absolutely essential to make testing an efficient and even workable effort. One repercussion of a haphazard, nondocumented test plan is that it does not allow for precise, repetitious testing. Therefore, ad hoc retesting cannot ensure that errors were in fact corrected. While the value of ad hoc testing cannot be completely dismissed during debugging, it cannot be substituted for formal, well designed test strategies.

Other than the realization that formal methods are necessary, there does not currently exist any single strategy to solve the testing problem. Software testing over the years has seen a number of different approaches, and several different strategies have evolved. It is important to note that manipulation of the input data and application of failure regions to the testing process is not a substitution or replacement for any testing strategy. Rather it is to be used in conjunction with a formal test plan that uses one or more of the established test strategies. The ultimate goal of the application of failure regions is to improve the test plan by eliminating unnecessary test cases and permitting detection of multiple faults during the formal testing process. Several of these strategies and their relevance to the concept of failure regions and manipulation of the input data are discussed below.

#### *(1) Walkthrough Testing*

Walkthrough testing is a human testing method which has found to be effective in detecting logic design and coding errors. In walkthrough testing the test participants act like computers and mentally execute the program with a small set of test cases. This small data set allows the testing individuals to implement failure region analysis on a limited basis while mentally executing the program, especially by applying boundary value analysis. When faults are discovered with an initial set of data, the tester can change the input data and determine if the new data set produces the same fault. This can provide the tester a hasty look at a part of the failure region and may give an indication of where the program fault is. This simple application of failure region analysis

during walkthrough testing can make implementation of this human testing process more efficient.

In 1978, a study conducted by Myers found that from 30% - 70% of the total faults found in a program were detected by code inspections and walkthroughs (Myers, 1979, p.19). However, it was recognized early that human testing was not nearly effective enough to be used in isolation. Although the human testing methods are still employed, they are usually utilized in conjunction with the more traditional computer based testing techniques (Myers, 1979, p.17).

## *(2) Mutation Testing*

In mutation testing, DeMillo, Lipton, and Sayward utilized a method known as program mutation in conjunction with the concept of coupling. Coupling involves the idea that simple errors are coupled to complex errors. The authors describe it as "the use of test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors." (DeMillo, Lipton, and Sayward, 1978, p.286) Program mutation is an interactive testing method that uses a measurement of the number and kinds of errors it is capable of uncovering as a determinant of the effectiveness of the test data selected.

While this approach was found to be effective in small programs, as with many other approaches, its utility in a large program is questionable other than in possibly identifying appropriate test data. It also suffers from the problem of infinity. The possible number of mutant programs in combination with a variable number of data

sets can be infinite and will therefore cause the same problem which was encountered with exhaustive testing - an infinite set of test cases.

Failure regions are just as applicable to mutation testing as they are to other test strategies. Mutation testing involves manipulation of both the program source code as well as the input data to evaluate the completeness of the test data. The use of failure regions affords the opportunity to skip redundant mutations or mutants that will always produce correct results.

### *(3) Module Testing*

Myers describes module testing as the process of testing the individual subroutines, packages or procedures in a program rather than testing the program as a whole. The purpose is to compare the function of a module to the specification of that module with the intent of showing that the module contradicts the specification. In module testing, the focus is on testing small blocks of the program. (Myers, 1979, p.77)

Again, module testing is only one aspect of the overall testing process and must be used in conjunction with other methods. Even if it were possible to exhaustively test each module, the program cannot be considered completely tested until it is tested as a whole entity, so that all modules are tested to ensure that all module interfaces work correctly.

Module testing makes use of failure region due to its white box nature. The tester initially derives test data after a review of a module's logic. The test is then executed on the module as it normally would be. The application of failure regions then allows the tester to refine the test data and conduct further testing of the



module's internal structure. This same process can be applied to each module as well as the interface between the modules.

#### *(4) Functional and Structural Testing*

Functional testing refers to the generation of test data to evaluate each specified function of the software. It assumes a black box approach in which implementation details are not important. On the other hand, structural testing is dominated by details. It refers to the generation of test data to evaluate each part of the structure of the software. While the two strategies differ in both outlook and purpose, there is no disagreement as to their use. While functional tests can, in theory, detect all errors in infinite time, structural tests are finite but cannot detect all errors (Beizer, 1983, p.5).

The fact that both strategies have recognized limitations and advantages allows a tester to incorporate the best of both strategies. Although there have been numerous studies which support a high error detection rate utilizing structural testing, there have been no definitive studies that show one method is better than the other in all situations. In 1976, Hetzel compared the fault detection capabilities of code reading, structural analysis, and functional testing. In this case the structural testing criterion used was statement coverage. His results showed that functional testing discovered the most faults, code reading discovered the least, and structural testing fell in between the other two. (Hetzel, 1976) Another study by Basili and Selby compared code reading by stepwise abstraction with functional and structural testing. The structural testing criterion used was again statement coverage. While code reading by stepwise

abstraction detected the most faults, structural testing detected the least number of faults.  
(Basili and Selby, 1987)

The key point here is that both of these strategies are relevant to the study of failure regions in that they both involve the analysis and manipulation of the input data. The major difference between these two testing strategies is the manner in which the input data is applied to the programs being tested. The analysis and identification of failure regions and their application in either of these strategies should greatly assist in refining the input data and making the testing more efficient - regardless of whether the structure of the program is being tested or whether the specified function is being tested.

#### *(5) Regression Testing*

Regression testing is the practice of repeating old tests after a change in code has been made to correct a fault. The purpose is to determine the effect of the old data on the corrected code. Additionally, repeated testing with the same data ensures that new faults have not been added during fault correction. Regression testing is a process that requires considerable resources. (Lamb, 1988, pp.116, 117)

The application of failure regions are particularly appropriate to regression testing. The failure region can be used as a guide in testing the correctness of the modified code. It also ensures that previous input data is available to conduct regression testing under the same conditions that the initial testing was performed.

#### *(6) Fault Sensitivity Analysis*

Fault sensitivity analysis is "the study of the propensity for a program to fail in the presence of faults." (Voas and Morell, 1989, p.1) In their work in the area of fault sensitivity, Voas and Morell formalize the idea that a program has a high probability of failure upon encountering a fault, that is, if it is difficult to hide a fault in the code, then the program is fault sensitive (Voas and Morell, 1989, p.1).

This concept of fault sensitivity parallels the study of failure regions. In their modeling of the process where execution of a fault leads to a failure, Voas and Morell describe three necessary preconditions: 1) a fault must be reached; 2) the fault must adversely affect the succeeding data state; 3) that effect must persist to the output (Voas and Morell, 1989, p.1). These preconditions correspond to the inequalities that bound the failure region being studied in this research.

Additionally, Voas and Morell assert that infection analysis suggests that certain locations in the program are fault-sensitive while others are fault-insensitive. They conclude that a fault-sensitive program requires more test data to ensure program correctness than a fault-insensitive program. (Voas and Morell, 1989, p.11) This idea parallels the idea of error clustering discussed earlier. The difference between fault sensitivity analysis and failure region analysis is that fault sensitivity analysis analyzes the program location and the distribution of faults in the code. This thesis analyzes the conditions for input to reveal faults in the code. While there is some duality in these two concepts, each application may be of separate usefulness.

Finally, while it is applicable to several other areas of software engineering, Voas and Morell's work is also applicable to the area of software testing. Specifically, they see infection analysis as appropriate to evaluating various test strategies. Although the infection analysis parallels the failure region analysis, failure region analysis is designed towards enhancing current testing strategies. Infection analysis can give an estimate of a program's complexity and, therefore, provide an indication of the test data adequacy.

*b. Benefits of failure region application*

The key issue in the previous discussion of the various testing strategies is that the incorporation of failure regions into those strategies can be highly profitable to the testing process. The proper application of failure regions allows the tester to avoid duplicate test cases while proceeding with other more appropriate test cases. This offers the additional benefit of detection of multiple errors prior to sending the program back for correction. Another key application is that failure regions can be incorporated with other conventional test strategies and is not intended to be used in isolation.

#### **D. OVERVIEW**

While there has been extensive research in the area of software testing, no testing methodology exists that ensures program correctness, especially correctness of large programs. There is a significant amount of current research aimed at examining the input data and its function in program testing, however, there has been very little research in the measurement and analysis of failure regions. This first chapter has provided an

introduction to the concept of failure regions and a summary of the ongoing research. The remainder of this thesis research will deal with manipulation of the input data and measurement and analysis of the shape and geometry of the failure regions.

Chapter II is a detailed description of the thesis methodology. It includes a description of the programs utilized in this thesis as well as a description of the specification from which the source programs were derived. Additionally, all assumptions and preconditions utilized during this research will be explained. The chapter describes in extreme detail the exact methodology used in manually defining the failure regions. This will be accomplished through a detailed analysis of the three conditions that define the failure region and the application of these three conditions to examples from the source code.

Chapter III is a presentation of the analytic results and a discussion and evaluation of these results. All observations made in the development of the failure regions will be presented as well as a discussion of all special cases. These observations are presented in light of their effect on the software testing process.

This is the focus of Chapter IV - a discussion of areas of potential research and all other conclusions and recommendations that have been derived from this research effort. And like previous research on program testing, that is the goal of this research - the improvement of the software testing process as well as the software development cycle and a foundation for further research.

## II. METHODOLOGY FOR ANALYZING THE FAILURE REGION

### A. SPECIFICATION AND PROGRAM SOURCE CODE DESCRIPTION

The source code programs analyzed during the course of this research are from a group of eight programs designed and written by eight pairs of undergraduate students from the University of California, Irvine. All student groups were provided the specification of a program called *Conflict* that simulated combat interaction between two armies.<sup>1</sup> The specification required each program to accept global data describing the position, size, and attributes of each army, plus a description of the terrain in which the armies conduct operations. A detailed list of the global declarations is depicted in Appendix A for reference purposes. The programs return a description of the encounters between the armies, plus the final condition of each army. (Shimeall, PhD Dissertation, 1989)

Each army is composed of one or more battalions, which are made up of one or more squadrons. The programs are given global data concerning information on the initial location and attributes of each battalion. Encounters between the opposing armies are based on the description of their individual battalions as well as the environmental conditions of weather and terrain. Battalions are able to perform five distinct functions: attrition, restoration, movement, communication and observation. The conflicts that occur

---

<sup>1</sup> In the text of this thesis, bold text refers to the conditions of a failure region, and italicized text refers to procedures, variables or statements from the source code.

in this combat simulation result from the interactions each battalion has with every other battalion in a given number of time intervals based on those five combat operations. The programs perform input and output only through global data structures. (Shimeall, PhD Dissertation, 1989)

Although the specification provided more detail than that indicated above, it was flexible enough so that the eight different versions of the program were varied in length, organization, structure and linearity. All programs were written in the PASCAL programming language and compiled using the UNIX Berkeley compiler. The number of lines of code in the programs varied from approximately 1,200 to 4,000. A major requirement was that each program had to compile, accept input, execute 15 sets of test data, and produce output for each input data set before it was accepted. After meeting these requirements, the number of faults discovered in each program varied from a low of 25 faults to a high of 50 faults. The variety of the programs in length, structure, and linearity provided an excellent data base from which to conduct a detailed analysis of failure regions, although only two of the eight available programs were used in this thesis. These particular programs were selected because one program had the most source code faults and the other had the least source code faults of the eight available programs. The intent was to obtain a somewhat divergent sampling of program quality.

## **B. ASSUMPTIONS AND PRECONDITIONS**

As with any research effort, there were certain assumptions made during the course of the development of the methodology of defining failure regions. This section points

out both the general assumptions allowed by this process as well as the more specific assumptions made concerning the particular programs used. The reader should be aware that all of the programs described above and utilized in this research meet these assumptions. The initial assumption regarding failure region analysis is that the programmers have conducted some low level testing and that the program can compile and execute. As stated above, each program in this research was required to compile and execute 15 input data sets. This ensured that the programs were at least marginally debugged by the program team and that the most obvious faults were removed. This is a realistic assumption since, in a real world software development process, the coding team would not send a program to the test team until the program compiled and executed at a minimum. This assumption, however, does not imply any degree of software reliability, either in a research or real world scenario.

It is assumed that acceptable and unacceptable output has been established prior to program testing. The establishment of acceptable output is necessary so that correct program behavior can be determined upon program execution. Additionally, when determining the conditions under which bad data is not masked (Condition III), it is necessary to know what is acceptable/unacceptable output so that it can be determined if output from the test runs is contaminated. This will have a significant impact on Condition III of the failure region. Furthermore, programs can be of any size and structure, although it will be shown later that structure has a significant impact on the level of difficulty of this process.



It is assumed that prior to the failure region analysis, the program has been tested using some formal test strategy. This supports a basic tenet of this research - failure region analysis is not designed to circumvent or replace testing efforts. Rather, it is designed to enhance the testing process through the refinement of test data. Once this test strategy has been implemented and faults have been identified and located, although not necessarily corrected, the failure region analysis can proceed. In this research, all programs were tested and faults were located using a variety of test techniques such as inspection, static analysis, code reading, etc. This also points out that the specific test technique used is not vital to the failure region analysis. The only consideration is that faults are identified.

It is assumed that uninitialized values are contaminated and, therefore, coincidental correctness does not occur. While this is in general difficult to support, as there are published accounts of correct behavior of uninitialized variables, this assumption obviates the need for very complex probabilistic arguments in the construction of failure regions. (Shimeall and Leveson, 1989, pp. 35, 37). This will be critical in establishing the conditions under which bad data values are not masked when defining the failure region. A further assumption is that uninitialized, improperly assigned, or improperly referenced pointers will be categorized and treated like uninitialized values - they are contaminated and are not assumed coincidentally correct.

It is assumed that during the process of developing the conditions that define the failure region, faults must be examined in isolation. This means that the conditions for each fault are developed without regard to other faults. This assumes that faulty code

does not have any affect on the conditions that define the failure region for another fault. This assumption exists strictly to simplify the analysis presented in this thesis and future work may involve setting it aside.

Finally, the failure regions presented throughout the course of this paper represent the unsimplified failure region. That is, the failure regions are not simplified to a reduced state. In an actual scenario, conditions AND'ed and OR'ed together would be logically simplified to their simplest state. Logical reduction or simplification is not done in this paper so that the failure region can be seen in its most complete state.

## **C. MANUAL METHODOLOGY FOR ANALYZING FAILURE REGIONS**

### **1. General**

The process of manually analyzing failure regions is not a particularly complex one, but it is labor intensive and time consuming, and it requires an in-depth knowledge of the source code. The goal of the failure region analysis is to make testing a more efficient process. At this early stage in the development of this manual process, the failure region analysis can improve the testing process but the advantage gained is offset by the time required to perform the analysis. To do this will require further refinement of the manual process and/or the automation of this process (Shimeall, FALTER, September 1989 and Shimeall, REACHER, September 1989).

Failure regions are bounded by the inequalities deriving the combination of three sources the reachability conditions for the code with the defect (Condition I); the conditions under which the calculations in that code produce an erroneous value

(Condition II); and the conditions under which the bad data isn't masked by later processing (Condition III) (Shimeall and Griffin, 1989, p.1). The logical AND of these three conditions derives the failure region. The absence of any or all of the three conditions implies a **TRUE** condition. This means that there is nothing in that condition that puts a limit on the failure region or nothing that precludes identification of the failure region. If any of these conditions prove impossible, the fault being analyzed will not cause a failure unless there is a change in the source code. The remainder of this chapter will be devoted to a detailed development of each of the three conditions that combine to form a failure region.

## **2. Development of the Reachability Conditions (Condition I)**

The reachability conditions for a fault refer to the set of conditions that allow the error producing code to be executed. The development of the reachability conditions for a particular fault is a matter of code reading. By necessity this step must begin at the start of the program. The goal is to identify all of those conditions that must be satisfied in order for the code with the defect to be reached. In order to accomplish this, the problem can be attacked in two phases. These two phases can be identified as the external reachability conditions and the internal reachability conditions. In the first or external phase, those conditions that are external to the procedure/function in which the fault occurs are established. In the second or internal phase, those conditions that are internal to the procedure/function in which the fault occurs are identified. Therefore, a failure region can have both external and internal reachability conditions, and the logical AND of these two sets forms the complete reachability condition.

To establish the external conditions, it is necessary to start at the beginning of the program. Appendix B is a segment of source code, the main procedure and two internal procedures taken from test program six to illustrate this process. This particular code has three identified faults, all in procedure *InitBattalion*: the first is located in the highlighted area between lines 30 and 31; the second fault is in the highlighted area between lines 52 and 53; and the third fault is located in the highlighted area between lines 54 and 55.

In this section of code, the main procedure, *Conflict*, first calls procedure *Initialize* which initializes the values of the *OldArmy*. The first line in this procedure does a check on the value of *Duration*. If the value of *Duration* is either negative or zero ( $Duration \leq 0$ ), the program will abort and produce an error message. Therefore, a reachability condition for any faults occurring later in the program and specifically for the three faults that occur in procedure *InitBattalion* is ( $Duration > 0$ ). Additionally, lines 099 through 110, depicted in Figure 2.1, are a series of checks to ensure that initialized variables are within acceptable ranges. If any of these variables fall outside of their acceptable ranges, the program will produce an error message and terminate. Therefore, the following reachability conditions are established by the checks in Figure 2.1:

```

099  if (NumWTypes < 0) or (NumWTypes > MaxWType) then
100    TellError ('NumWTypes is out of possible range');
101  if XDelta <= 0 then
102    TellError ('XDelta is not greater than zero');
103  if YDelta <= 0 then
104    TellError ('YDelta is not greater than zero');
105  if (NumWEvents < 0) or (NumWEvents > MaxWeather) then
106    TellError ('NumWEvents is out of possible range');
107  if SampleRate < 0 then
108    TellError ('SampleRate is negative');
109  if IMeanAlt <= 0 then
110    TellError ('IMeanAlt is not positive');

```

**Figure 2.1 Range Checks on Initialized Variables**

```

((NumWTypes >= 0) AND (NumWTypes <= MaxWType))
    AND
    (XDelta > 0)
    AND
    (YDelta > 0)
    AND
    (NumWEvents >= 0) AND (NumWEvents <= MaxWeather))
    AND
    (SampleRate > 0)
    AND
    (IMeanAlt > 0)

```

Lines 112 through 120 depicted in Figure 2.2 also establish a distinct set of reachability conditions that must be satisfied before the program can proceed. These conditions may be expressed in the following terms:

```

((WEvent  $\in$  [1..Params.NumWEvents])
  AND
  (Params.NumWEvents > 0))
  AND
  (TStart <= TEnd)
  AND
  (WRadius > 0)
  AND
  (WSeverity <= Params.WMaxSeverity)

```

```

112  for WEvent := 1 to Params.NumWEvents do
113    with Weather[WEvent] do

114      begin
115        if TStart > TEnd then
116          TellError ('Bad starting and ending time for weather event');
117        if WRadius <= 0 then
118          TellError ('Bad radius for weather event');
119        if WSeverity > Params.WMaxSeverity then
120          TellError ('WEvent Severity > Params.
              WMaxSeverity');

```

**Figure 2.2 Requirements Defining the Reachability Conditions**

Finally, lines 122 and 123 shown below also establish yet another set of reachability conditions that must be satisfied.

```

122  for Side := false to true do
123    for Batt := 1 to NArmy[Side] do

```

These conditions can be expressed in the following terms:

$$\begin{aligned}
& ((\text{Side} \in [\text{false}] \mid \text{NArmy}[\text{false}] > 0) \\
& \quad \text{AND} \\
& \quad (\text{Batt} \in [1..\text{NArmy}[\text{false}]]) \\
& \quad \text{AND} \\
& ((\text{Side} \in [\text{true}] \mid \text{NArmy}[\text{true}] > 0) \\
& \quad \text{AND} \\
& \quad (\text{Batt} \in [1..\text{NArmy}[\text{true}]])
\end{aligned}$$

The logical AND of the above conditions are summarized in Figure 2.3 and establish the external reachability conditions for the three faults in Appendix B.

Now that all external conditions have been satisfied, the conditions internal to the function/procedure in which the fault occurs must be checked. In this case, the three faults are located in procedure *Initialize*, the procedure which checks for input correctness. Again, it is necessary to identify all of those conditions that must occur in order for the program to reach line 30, the last line of code prior to the location of the first fault. The first check is at line 21 and involves the line of code:

021    *for WeaponType := 1 to Params.NumWTypes do*

Obviously, *WeaponType* must be greater than 0 and less than or equal to *Params.NumWTypes*. Therefore, the reachability condition for fault number one also includes the internal conditions:

$$((\text{WeaponType} \in [1 \dots \text{Params.NumWTypes}]) \text{ and } (\text{Params.NumWTypes} > 0))$$

Since there are no other internal conditions that must be established in order to reach the faulty code, these conditions must be logically AND'ed with the external conditional requirements established earlier. As a result, the complete reachability conditions for the first fault are shown in Figure 2.4.

```

        (Duration > 0)

        AND

        (((NumWTypes >= 0) AND (NumWTypes <= MaxWType))
         AND
         (XDelta > 0)
         AND
         (YDelta > 0)
         AND
         (NumWEvents >= 0) AND (NumWEvents <= MaxWeather))
        AND
        (SampleRate > 0)
        AND
        (IMeanAlt > 0))

        AND

        (((WEvent ∈ [1..Params.NumWEvents])
         AND
         (Params.NumWEvents > 0))
         AND
         (TStart <= TEnd)
         AND
         (WRadius > 0)
         AND
         (WSeverity <= Params.WMaxSeverity))

        AND

        (((Side ∈ [false] | NArmy[false] > 0)
         AND
         (Batt ∈ [1..NArmy[false]]))
         AND
         ((Side ∈ [true] | NArmy[true] > 0)
          AND
          (Batt ∈ [1..NArmy[true]]))))

```

**Figure 2.3 External Reachability Conditions**



```

        (Duration > 0)

        AND

        (((NumWTypes >= 0) AND (NumWTypes <= MaxWType))
         AND
         (XDelta > 0)
         AND
         (YDelta > 0)
         AND
         (NumWEvents >= 0) AND (NumWEvents <= MaxWeather))
        AND
        (SampleRate > 0)
        AND
        (IMeanAlt > 0))

        AND

        (((WEvent ∈ [1..Params.NumWEvents])
         AND
         (Params.NumWEvents > 0))
         AND
         (TStart <= TEnd)
         AND
         (WRadius > 0)
         AND
         (WSeverity <= Params.WMaxSeverity))

        AND

        ((WeaponType ∈ [1..Params.NumWTypes])
         AND
         (Params.NumWTypes > 0))

```

**Figure 2.4 Reachability Conditions (Condition I)**

The next fault, located immediately after line 52, presents a different situation. The external requirements exist just as they did for the first fault. Obviously, this must hold true in order for the procedure *InitBattalion* to be executed. However, in this case, there are no other conditional statements that affect the execution of the faulty code. As a result, the only conditions that must be established and the complete reachability conditions for fault number two are again depicted in Figure 2.3.

The same situation exists for fault number three. The external requirements still exist, but there are no other conditions that must be met in order for the faulty code to be reached. Therefore, the internal reachability condition is **TRUE**, and once again, the only reachability conditions for fault number three are the external reachability conditions depicted in Figure 2.3.

This same process will establish the reachability conditions for any faults located and identified in the source code. Again, the key point in this stage of the process is to identify those conditions in the source code prior to the faulty code that must be met in order for the faulty code to be executed. The logical AND of these conditions is the reachability condition (Condition I) for the failure region for that particular fault.

### **3. Development of the Error Generation Conditions (Condition II)**

The error generation conditions (Condition II) are those specific conditions that cause the fault to be executed in a way that an erroneous intermediate result or error is produced. If the error is generated under all conditions, or rather, the error will be generated every time the code is reached, then the error generation condition is **TRUE**.

If the error cannot be generated, the error generation condition is **FALSE** and the failure region analysis ceases.

The source code in Appendix C will be used to demonstrate the development of the error generation condition. In this particular code, the first fault, located immediately after line four, is a missing check on the value of *OldArmy[DestArmy, Dest].Squadrons*, i.e. a piece of missing code. In this case, as long as *OldArmy[DestArmy, Dest].Squadrons* > 0, there is no fault. However, if this value is less than or equal to 0, incorrect operations may be performed on destroyed battalions. Therefore, the error generation condition for this fault is:

$$(\text{OldArmy}[\text{DestArmy}, \text{Dest}].\text{Squadrons} \leq 0)$$

The next fault is located immediately after line five and involves the incorrect use of the variable *CommandsFinished^.msg*. The correct code and incorrect code are shown in Figure 2.5. In this example, the fault will not generate an error as long as *CommandsFinished^.msg* = *Cmsgs[DestArmy, msg].msg*. Conversely, the fault will generate an error whenever *CommandsFinished^.msg* <> *Cmsgs[DestArmy, msg].msg*. Therefore, the error generation condition for this fault is:

$$(\text{Cmsgs}[\text{DestArmy}, \text{msg}].\text{msg} \neq \text{CommandsFinished}^{\wedge}.\text{msg})$$

The final fault in this code segment is located immediately after line 14. The fault in this case is that *NewArmy* is not updated to match *OldArmy* after command messages are implemented. This fault is similar to the first fault explained for the error generation conditions, that is, it involves a missing piece of code:

$$\text{NewArmy}[\text{DestArmy}, \text{Dest}] := \text{OldArmy}[\text{DestArmy}, \text{Dest}]$$

```
CheckBattalionConstants (Cmsgs[DestArmy, msg].msg,  
    Params.NumWTypes);  
Army[DestArmy, Dest] := Cmsgs [DestArmy, msg].msg;
```

a) Correct code

```
CheckBattalionConstants (msg, Params.NumWTypes);  
Army[DestArmy, Dest] := msg;
```

b) Incorrect code

**Figure 2.5 Source Code Example**

This fault will occur every time that it is reached, therefore, the error generation condition for this fault will again be **TRUE**.

The three examples presented above represent the error generation conditions (Condition II) for three distinct faults. To define their respective failure regions, these conditions must be logically AND'ed with their respective reachability conditions (Condition I) and the conditions under which the errors are not masked by later processing (Condition III).

#### **4. Development of the Conditions Under Which the Fault Is Not Masked (Condition III)**

The conditions under which a bad value or values are not masked by later processing define the third boundary of the failure region (Condition III). Again in this case, it is necessary to know the exact location of the fault in the source code. Once the

fault is identified and located and the two previous conditions are established, the next step is to determine the conditions under which the bad data cannot be masked by further processing. This is done by determining what the contaminated data is and determining where and how the bad data is next used. This must be done until the data is used to generate output to determine whether or not the bad data is masked prior to program output. In the example programs utilized in this research, the only output occurs at the completion of program execution. It appears the easiest way to do this is to determine all of the conditions that may cause the bad data to be masked and then negate these conditions. The two ways in which a bad value is masked by further processing are: a) the bad data can be overwritten by an acceptable data value; or b) the bad data never contributes to the final result.

The source code in Appendix D will be used to demonstrate the development of Condition III. Again, when establishing the failure region, Condition I and Condition II should be resolved prior to determining Condition III. Also, as in developing the reachability conditions, there are both internal and external conditions that must be satisfied. The main task in developing Condition III is to trace the program from the fault and determine what variables become contaminated and how these variables affect the output. In the case of the fault in Appendix D, the result of the fault is the contamination of the variable *Restoration[Sqd, 0]* in line ten. This condition is also internal to the procedure in which the error occurs. Examining the source code we observe that the variable *Restoration[Sqd, 0]* is assigned inside of the for loop starting at line seven:

*for Sqd := 1 to Army[Armynum][Batt].Squadrons do*

Since the assignment statement is inside of a loop, this should be a flag indicating that it is possible that either a good or bad value of *Restoration[Sqd, 0]* could be overwritten by another value during later processing. Therefore, the internal conditions under which a bad value of *Restoration[Sqd, 0]* could be overwritten and masked are:

$$\begin{aligned} & ((\exists (s) \mid (s > sqd) \\ & \quad \text{AND} \\ & \quad (s \leq \text{Army}[\text{Armynum}][\text{Batt}].\text{Squadrons})) \\ & \quad \text{AND} \\ & \quad (\min(\min(\text{FixRate} * \text{NumFixers/Casualties}, \text{FixSuppl/Casualties})), \\ & \quad \text{Endurance0[Sqd]} - \text{Endurance[Sqd]}) \leq \text{Restoration[Sqd, 0]}) \end{aligned}$$

This equation translates into some later value of *s* which allows a good value of *Restoration[Sqd, 0]* to overwrite and mask a bad value. This condition must then be negated to arrive at the true conditions that do not allow a value to be masked by later processing (Condition III).

## 5. Example of Defining the Entire Failure Region

The preceding sections have provided the details as how to specifically define each individual condition of the failure region. The following example is a summary of those examples and will provide an example of defining all three failure region conditions for a single fault. The source code in Appendix E will be used for this example.

This particular code segment involves the main procedure of *Conflict* and one sub procedure (*InitVals*). The fault occurs immediately after line 32. The fault is that the initial *Velocity* is assigned incorrectly in line 33 if *Endurance*  $\leq 0$ . Since *InitVals* is the first procedure called and it is also the first line in the main procedure, there are no external conditions that must be satisfied before *InitVals* is reached. The next step is to

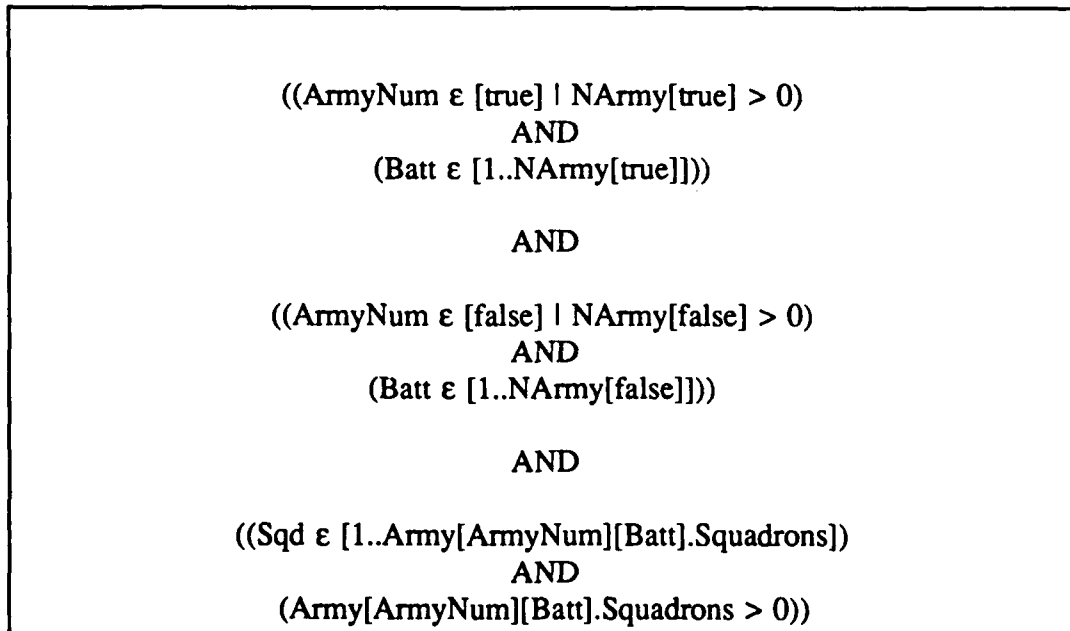
determine the internal reachability conditions. The conditionals in the statements below include the code where the fault occurs and are good candidates for reachability conditions.

```

003 for ArmyNum := false to true do
004   for Batt := 1 to NArmy[ArmyNum] do
...
031     for Sqd := 1 to Army[ArmyNum][Batt].Squadrons do

```

After further analysis, these 'for' statements produce the internal reachability conditions depicted in Figure 2.6 which must be satisfied before the faulty code is reached. Since there are no external conditions, Figure 2.6 also represents the complete reachability conditions for the fault.



**Figure 2.6 Internal Reachability Conditions**

The error generation conditions (Condition II) for this fault occur when  $Army[ArmyNum][Batt].Endurance[Sqd] \leq 0$  and when this velocity ( $VO$ ) is less than the previous value of *Velocity*. This can be expressed by the following error generation conditions:

$$\begin{aligned}
 & (Army[ArmyNum][Batt].Endurance[Sqd] \leq 0) \\
 & \quad \text{AND} \\
 & \quad (Army[ArmyNum][Batt].VO[Sqd] < y \\
 & \quad \quad \text{where } y \in [x] \mid x = 9999 \text{ OR} \\
 & \quad \quad (\exists (s) \mid 1 \leq s \leq Sqd - 1), \\
 & \quad \quad x = Army[ArmyNum][Batt].VO[s])
 \end{aligned}$$

The final conditions of the failure region are established by determining the conditions under which a bad value for *Velocity* cannot be masked. This happens internally in *InitVals* when a subsequently good value of  $VO[Sqd]$  overwrites a bad value in lines 30 - 32. This derives the following internal conditions under which a bad value cannot be masked by further processing (Condition III):

$$\begin{aligned}
 & ((\exists s \mid (s > Sqd) \\
 & \quad \text{AND} \\
 & \quad (s \leq Army[ArmyNum][Batt].Squadrons)) \\
 & \quad \text{AND} \\
 & \quad (Army[ArmyNum][Batt].VO[s] \Rightarrow Velocity))
 \end{aligned}$$

The external conditions are those non-masking conditions external to the procedure that contains the fault. In this case, although the contamination of the variable *Velocity* in line 33 causes the contamination of several other variables in external procedures, there are no further external conditions under which the value of *Velocity* is not masked by later



processing. Thus, the failure region is defined by the logical AND of the three sets of conditions (Condition I, II, and III) just specified and is shown in Figure 2.7.

#### **D. CONCLUSION**

This chapter has provided a detailed look at the process of manually developing and the three conditions that define software failure regions. Additionally, Appendix F contains a list of the failure regions analyzed in this study. While this chapter has provided an examination of how the process works, the next chapter is an analysis of that process - the advantages, the disadvantages, and its application to the overall software engineering process.

**(Condition I)**

((ArmyNum  $\in$  [true]  $\mid$  NArmy[true] > 0)  
AND  
(Batt  $\in$  [1..NArmy[true]]))  
AND  
((ArmyNum  $\in$  [false]  $\mid$  NArmy[false] > 0)  
AND  
(Batt  $\in$  [1..NArmy[false]]))  
AND  
((Sqd  $\in$  [1..Army[ArmyNum][Batt].Squadrons])  
AND  
(Army[ArmyNum][Batt].Squadrons > 0))

**(Condition II)**

(Army[ArmyNum][Batt].Endurance[Sqd] <= 0)  
AND  
(Army[ArmyNum][Batt].VO[Sqd] < y  
where y  $\in$  [x]  $\mid$  x = 9999 OR  
 $\exists$  (s)  $\mid$  1 <= s <= Sqd - 1,  
x = Army[ArmyNum][Batt].VO[s])

**(Condition III)**

(( $\exists$  s  $\mid$  (s > Sqd)  
AND  
(s <= Army[ArmyNum][Batt].Squadrons)  
AND  
(Army[ArmyNum][Batt].VO[s] >= Velocity))

**Figure 2.7 The Complete Failure Region**

### **III. OBSERVATIONS AND FINDINGS**

#### **A. INTRODUCTION**

The goal of identifying the failure region for a detected fault is to determine which further input data test cases fall within the failure region. Those recognizably redundant test cases may be set aside and testing may proceed with the remaining test cases. After fault correction the failure regions can be used as a guide in testing the corrections to the code (Shimeall and Griffin, 1989, pp. 1,2). This chapter presents observations about the failure region analysis process and about the individual failure regions. It is quite possible that some of these ideas may be profitable in the automation of this process or in other testing research.

This research is explorative in that it is the first in-depth look at the manual analysis for software failure regions. This study was primarily devoted to the development and understanding of the failure region analysis process itself. It is anticipated that future work will empirically explore characteristics of the regions and the information they yield about faults in computer software. While satisfying the primary goal some interesting factors of the failure regions were observed, and these factors are also described in this chapter. However, most of the observations will focus on the process of failure region analysis. To focus the development of failure region analysis, this research was conducted using a single application. That is, the research data base was a single program specification and a set of two programs built according to that single

specification. While it is believed that these observations are applicable in a larger context, that belief is not yet supported empirically. As such, the reader is urged to use caution in applying the results below to different applications.

During the process of manually developing the failure region, it is necessary to view each fault in isolation. This means that the conditions for each failure region determined from the code in its uncorrected form. The analysis of this process should not attempt to alter any conditions of that region based on a perception of another fault, a fault correction, or another failure region. This is because this effort is not a debugging process - it is a means of reducing test effort by determining redundant test data. Failure region analysis makes no assumptions about how faults may be corrected, and faults do not need to be corrected prior to analyzing for failure regions.

## **B. OBSERVATIONS**

### **1. FAILURE REGION OBSERVATIONS**

By definition, a failure region consists of the logical AND of three distinct subregions, and it is precisely these three conditions that must occur in order for the fault to be reached, the error to be generated, and the bad data to be carried through to the output. If the input data varies from any of the three conditions of the failure region, then the failure region will be negated because one of the following three things will occur: a) the fault will not be reached; b) the error will not be generated; c) the bad data will be masked prior to program output or will not contribute to the final output. Therefore, it is only the three precise conditions of the failure region for a specific fault that will

allow that fault to occur. Any variance of these conditions will either cause a different error to occur or will cause no error.

One observation that can be made about the software failure regions examined in this research is that the failure regions tend to consist of disjoint subregions. That is, individual failure regions generally consist of distinct conditions that delimit disjoint portions of the data input space. While there may be some overlap, this was the exception rather than the rule.

The geometry of failure regions supports the concept that software failure regions may contribute to a more efficient testing process by removing redundant test cases. Since the sub regions tend to be disjoint and often in non-trivial ways, each failure region defined may identify subtly related portions of the input data space that can be eliminated in future test cases until faults are corrected. The size of the failure region and its subregions is not normally a function of program size, but more a function of the number of variables in its boundary conditions.

A significant effort was made in trying to draw some conclusions about the geometry of a failure region based on the type of error, i.e. infinite loops, initialization faults, etc. While it may be possible to draw some conclusions about how a certain type of error affects an individual region boundary condition, the fact that each failure region is delimited by three separate and distinct conditions makes generalizing about the overall failure region virtually impossible at this point. One general observation is that the type of error does seem to have an impact on the error generation condition (Condition II), however, trying to draw more precise conclusions about this idea is premature at this

point. It is hoped that further research using a broader failure region data base will permit general observations about the overall shape of the regions based simply on error type.

## **2. PROCESS OBSERVATIONS**

### ***a. General***

The manual process of defining software failure regions is not particularly difficult to comprehend, however, it is a time-consuming and laborious procedure to implement. The level of difficulty is dependent on both the length of the program and its components as well as the program complexity. This process is much easier to apply to a linear, non-nested program. In the case of a nested program, establishing the three failure region conditions becomes increasingly difficult as the nesting increases. In the case of the Pascal programming language, the conditionals *if...then*, *if...then...else*, *for...do*, *while...do*, *repeat...until*, *case...of* establish the program nesting that increases the difficulty of defining the failure regions. Planned automation of the region analysis process should alleviate much of this difficulty.

### ***b. Comparative Difficulty of Determining Failure Region Conditions***

Beginning with the most obvious findings, the first result that became clear during the course of analyzing for the failure regions was the comparative difficulty or ease of determining the various conditions of the failure region. It quickly became apparent that establishing the reachability and error generation conditions was the easiest portion of the analysis. As explained in the previous chapter, establishing the reachability conditions is simply a matter of determining what conditional statements contain the

erroneous code and then determining what conditions allow that code to be reached. Those conditional statements that do not affect the erroneous code do not even need to be considered. This code reading approach is quite simple and does not present any unusual problems in the identification of the reachability conditions.

Also, the identification of reachability conditions for one fault will frequently assist in determining the reachability conditions for another fault. This is particularly true when the faults are located in close proximity to each other or lie along the same paths. Figure 3.1, extracted from Appendix D, is the main procedure from source program number three and provides a good example of this situation. Lines 11 and 12 are pieces of code that were missing from the original source code. The original error is an inaccurate assumption that *Duration* = 0 is an erroneous value. Therefore, the condition **Duration** >= 0 becomes a reachability condition for any fault that occurs later in the program and must be included as a reachability condition in all other failure regions.

The error generation condition is also a fairly easy condition to establish in terms of relative difficulty. Once the fault is identified, establishing Condition II conditions is simply a matter of determining the specific conditions that cause the error to occur. Since it doesn't involve tracing through the program source code, it can frequently be easier to determine than the reachability conditions (Condition I). Again, as stated earlier, it is not actually necessary to distinguish between Condition I and II conditions when the distinction is vague. This holds true as long as the condition is addressed in the failure region definition. However, special care must be taken to ensure

```

001 begin          { Conflict}
002   InitVals;
003   if Duration > 0 then
004     for Mainloop := 0 to Duration do

005       begin
006         DoAction (Mainloop);
007         UpDate (Mainloop);
008         Output (Mainloop);
009       end
010   else

*****

011     if Duration = 0 then
012       Output(0)

*****

013   else
014     TellError ('Invalid Duration value');
015 end;          { Conflict}

```

**Figure 3.1 Example of Duplication of Reachability Conditions**

a complete statement of the conditions that lead to an error being generated by a fault in the program source code.

The establishment of the conditions in which the bad values resulting from a fault aren't masked by later processing is by far the most difficult to establish. These conditions have both an internal and external dimension. Additionally, the program must be traced through from the occurrence of the error to the point in the program where the contaminated data may be output. While one variable may be contaminated initially,



this variable can contaminate many other variables. These other contaminated variables must be traced through to program output. As the contaminated variables are traced, all masking conditions must be established and then negated to determine the Condition III conditions. In the case of a large program and significant variable contamination, this can be a very time consuming and difficult process.

In the course of this study, for the average failure region, the establishment of Condition I required approximately 35% of the time, Condition II required approximately 15% of the time, and Condition III required approximately 50% of the total time. The reader is cautioned that these figures are both relative and approximate. Also, this study started after all of the faults were identified. The time required to identify and localize a fault in a program may add substantially to the time required to establish Condition II. However, localizing and identifying faults is a required part of normal development and this work assumes that it has been completed prior to starting the failure region analysis process.

*c. Common Causes for Generating TRUE Conditions*

There are several situations that exist within the program source code that allow general statements to be made about the conditions of a failure region. These are situations that cause a specific condition of the failure region to be **TRUE**. These special circumstances dictate that: a) a fault will always be reached or; b) an error will always be generated or; c) there are no conditions in which the bad value is not masked by later processing. A short description of each of the situations and how they apply to the failure region conditions is provided below.

### (1) *Error Reachability Conditions*

If there are not any conditionals that affect the erroneous code, i.e. there are no conditions affecting the reachability of the erroneous code, then the reachability condition will always be **TRUE**. This means that there must not be any *if...then*, *if...then...else*, *for...do*, *while...do*, *repeat...until*, *CASE...of* statements anywhere in the flow of execution prior to the occurrence of the fault that prevent the execution of the erroneous code. This must hold true both internally and externally, that is, in both the procedure/function in which the fault occurs and in the other procedure/functions that were executed previously.

### (2) *Error Generation Conditions*

The situations in which the error generation condition will always be **TRUE**, i.e. the error will always occur, are more difficult to make general conclusions about. Generally, the error generation conditions will be **TRUE** under all circumstances except in those cases where the erroneous code involves a decision that allows the source code to behave in more than one way or when there has been an incorrect substitution of one variable or constant for another variable or constant. The first situation usually involves the reserved words *AND*, *OR*, *if...then*, *if...then...else*, etc that cause a program to do one thing under certain conditions and something else under a different set of conditions. The second situation involves a mistaken substitution of variables or constants. This is particularly true in the case of calculation or assignment statements and equality/inequality statements.

### *(3) Conditions Under Which a Bad Value Isn't Masked by Later Processing*

The situations that cause Condition III to be **TRUE** are when there are no specific conditions that do not allow a bad value to be masked by later processing. This occurs whenever there are no conditions that allow a bad value to be masked after the occurrence of the error. While it is difficult to make general statements about the circumstances that always cause Condition III to be **TRUE**, Condition III will always be **TRUE** whenever the program terminates prematurely. This is the most common situation under which Condition III will always be **TRUE**. An example of this situation would be in the case of a divide by zero error that causes the program to terminate. This will generate a **TRUE** condition in all cases. Another general situation that generates a **TRUE** condition is when faults occur in the generation of the output. This too will always cause a **TRUE** condition.

#### *d. Common Ground Between Failure Region Conditions*

As explained in Chapter II, both Condition I and Condition III conditions have an internal and external aspect. The internal aspect refers to the procedure/function in which the erroneous code is located while the external aspect refers to those procedures/functions outside of the ones that contain the erroneous code. This can have significant repercussions since the external conditions can and generally will impact on both the size and dimensions of the failure regions.

The fact that Condition I and Condition III conditions have an internal and external aspect offers an interesting sidelight. As has been pointed out earlier, the

development of the Condition III external conditions is the most difficult and time consuming part of this process. It may be possible to speed this process by examining the complete conditions for Conditions I and II and only the internal conditions for Condition III. Prior to this being done, however, more research needs to be done in order to determine the extent of the data input being reduced by each condition. In terms of relative time required to establish Condition I, establishing the external reachability conditions takes approximately 70% of the total effort as opposed to 30% to establish the internal conditions. In regards to Condition III, establishing the external conditions takes approximately 75% of the time while establishing the internal conditions takes only about 25% of the time. These relative times can only be approximations since program length, organization, and complexity can also significantly impact the establishment of the failure region conditions. It is conceivable that significant portions of the input data space can be reduced by looking selectively at parts of the failure region while reducing the time that this manual process actually takes.

During manual analysis, we often found that the distinction between whether a condition is a reachability condition (Condition I) or an error generation condition (Condition II) can be ambiguous. The distinction between an error generation condition (Condition II) and the condition under which a bad value cannot be masked by further processing can be equally vague. In the manual process, it is not crucial to make this distinction as long as the ambiguous condition(s) is addressed in the failure region under one of the conditions. However, this also suggests that the conditions that are being used to define the failure region may not be completely appropriate. Since there

is overlap between Condition I and Condition II and between Condition II and Condition III, it may be possible that we need to reexamine the basic definition of a failure region in terms of the three conditions. In other words, this ambiguity suggests that future research may find other ways of defining a failure region.

*e. Variable Contamination*

The contamination of variables is another significant aspect of this process. A variable is considered contaminated when it is modified by the erroneous code or uses a result of the erroneous code. While this is certainly source code dependent, variable contamination in the programs utilized during the course of this research propagated rapidly (certainly a relative term that is difficult to define with such a limited empirical base). In one case, a single contaminated variable caused the subsequent contamination of ten other variables in subsequent procedures. This affects the analysis of the conditions in which a bad value isn't masked by later processing. Since it is necessary to trace the contaminated variables through until either they are overwritten or output, the level of contamination is a considerable factor. However, while variable contamination was considered high during this process, this contamination was generally vertical as opposed to horizontal. That is, variables generally contaminated other variables in a linear fashion one after another as opposed to the contamination of several variables in a tree-like fashion. This is important because it tends to limit the number of variables that need to be traced through the program at exactly the same time. A vertical contamination also tends to produce a failure region that is smaller in size and dimension than a horizontal contamination.

#### **IV. CONCLUSIONS AND RECOMMENDATIONS FOR FUTURE RESEARCH**

In addition to increasing the efficiency of testing, the process of analysis for failure regions provides several advantages during software development. The analysis of failure regions provides some detailed information about the source code. It provides a detailed look at the organization and structure, complexity, and flow of the program.

In particular, the process highlights the connections between various variables in the program and the propagation of their values. This may suggest useful ways that the program may check its internal state during execution. It also offers a closeup view of the program faults - where they occur, along what paths they occur, and possibly some ideas on the clustering of faults in the source code. If rewrites are needed, the localization of faults provides a basis for making intelligent decisions based on where the efforts are best spent.

Furthermore, failure region analysis also gives information about the application requirements. The failure region process requires a closeup analysis of the input data. This analysis allows the user to compare data values to the requirements and analyze consistency between the requirements and the actual code. This is particularly true in the case of data boundary conditions.

Failure region analysis provides the tester with information about the testing process. In particular, failure region analysis provides a means of differentiating between anomalies and faults detected by previous testing techniques. Anomalies are abnormal

constructs detected by either manual or automated static analysis techniques. These abnormal constructs may either be faults or constructs that cause either intentional or unintentional side effects. Failure region analysis may be used to determine if an anomaly is a fault. As pointed out earlier, this process also provides a means of guiding regression testing to make regression testing a more efficient process. Finally, failure region analysis provides the tester feedback on the quality of the input data selected to be part of the input data test set.

One factor that may reduce the cost of the failure region analysis process is its cross-application with other testing techniques. The manual failure region process as defined herein requires a detailed analysis of the source code. As such, it is possible that it could be used in conjunction with other techniques that require a similar analysis. For example, reachability analysis is part of structural testing and error generation analysis is part of code inspections or code reading. It is quite possible that each of these processes and others not specified here could be used in conjunction with each other, thereby gaining the benefits of each process while minimizing the costs.

## **A. CONCLUSIONS**

This thesis has been the first in-depth examination of the manual development of software failure regions. The observations and findings presented in the previous chapters provide initial basic knowledge about the process. As pointed out in Chapter III, there are limitations on this research. Despite these limitations, it has provided an excellent

basis from which some guardedly general conclusions can be drawn and it is an ideal source for future research topics.

While this research has identified the difficulty of implementing this process as a manual tool, the automation of this process will significantly enhance the testing process. It will make initial testing more efficient, and it will also improve the process of regression testing. However, even when this process becomes fully automated, the problem of manipulating and statistically analyzing multi-dimensional failure regions must be solved. This involves improvements in theoretical understanding of the relationships between  $n$ -dimensional objects and is the subject of ongoing research.

## **B. RECOMMENDATIONS FOR FUTURE RESEARCH**

There are several areas of research that are logical follow-ups to this work. Several of these involve the idea of statistical analysis of the failure regions. The first area that needs to be addressed before this concept can be utilized profitably is the capability to statistically analyze multi-dimensional failure regions. A critical aspect of this idea is the ability to be able to graph these regions and draw conclusions about the shape and geometry of the software failure regions. The ability to do statistical analysis on the failure regions may provide information on the clustering of failure regions. Another aspect of this statistical analysis is the ability to determine the size of the failure regions relative to the total data input space. This will allow the testers to derive a mathematical estimate of the percent of the data input space that can be eliminated from further testing. Another beneficial aspect of the statistical analysis is the ability to be able to determine



how much each condition of the failure region contributes to the overall reduction of test cases.

There are several other areas of potential research that involve software failure region analysis. The idea of applying failure region analysis earlier in the software engineering life cycle process is an open question. It is entirely feasible that failure region analysis can be applied during the development of the requirements and design as a means of reviewing the consistency, sufficiency, feasibility and testability of the system during development.

Another area of potential research is the examination of the estimation of cost of error correction. This is the process of estimating the expense of correcting faults in the source code that result in errors. Empirical data must be developed to determine if the cost of error correction is greater than the benefit derived and how this cost factor is influenced by the number, size and dimensions of the subregions. It may be determined that in some cases the cost of error correction outweighs the benefits of fixing the fault, therefore, the fault is identified and commented but not corrected.

This study was based entirely on a data base that executed numerical data input. Further study is needed to determine the feasibility of applying the failure region analysis to programs that execute non-numeric input data. While it is believed that this process is applicable to this type of program, this has not been established empirically. Additionally, more research is needed to determine the applicability of this process to other applications. One particular example is the application of this process to production-quality code. Empirical data is needed to determine if this process is easier

if applied to well tested code or more difficult. It would be expected that unit tested, non production-quality code would contain more faults and failure regions, while the faults in production-quality code would be more subtle. While the subtlety of the faults will have a greater impact on the fault detection technique used in the testing process, further research is needed to determine if the geometry of a failure region differs between production and non production-quality code.

Finally, further study must be done to establish the limitations of this process. Of particular concern is the derivation of the external conditions of Condition I and Condition III in large programs. As a result of this study, it is recognized that this process certainly becomes more difficult as program size increases. However, the rate of increase of effort still needs to be established empirically.

The analysis of software failure regions is a relatively new field of study, and there are many open questions. This research has established some initial observations, however, it has opened up many more questions to be explored in the near future.

## APPENDIX A

This Appendix lists all of the global PASCAL declarations utilized in the source code. These terms are referred to in the thesis text and appendices. (Shimeall, PhD Dissertation, 1989)

### CONSTANTS

```
MaxBattalion = 5;  
MaxMsg       = 10;  
MaxTerrain   = 50;  
MaxSquadron  = 50;  
MaxWeather   = 50;  
MaxType      = 50;
```

### TYPES

```
Results      = (None, Observed, Engaged);
```

```
TYPE WRec    = RECORD
```

```
    NumWeapon : INTEGER;  
    UseLimit   : INTEGER;  
    Damage     : REAL;  
    Range      : REAL;  
    Radius     : REAL;  
    FireRate   : REAL;  
END;
```

```
WEvent       = RECORD
```

```
    TStart, TEnd : INTEGER;  
    WXO, WYO, dWX, dWY : REAL;  
    WRadius, WSeverity : REAL;  
END;
```

```
PRec         = RECORD
```

```
    ISlopeFactor, IAltFactor, IMeanAlt, IX, IY, IC : REAL;  
    NumWTypes : INTEGER;  
    XDelta, YDelta : REAL;  
    WMaxSeverity : REAL;  
    NumWEvents : INTEGER;  
    MaxPriority : INTEGER;  
    SampleRate : INTEGER;  
END;
```

```

Battalion      = RECORD
    CommJamEff : REAL;
    CommJamRadius : REAL;
    CommJamPriority : ARRAY [1..MaxBattalion] OF INTEGER;
    Endurance : ARRAY [1..MaxSquadron] OF REAL;
    FixRate : REAL;
    FixSuppl : REAL;
    GRow : INTEGER;
    MaxSlope : REAL;
    MediaDelay : REAL;
    MWEffect : REAL;
    NumFixers : INTEGER;
    NumJammers : INTEGER;
    NumProcess : INTEGER;
    NumReceive : INTEGER;
    NumSend : INTEGER;
    ObsJamRadius : REAL;
    ObsJamEff : REAL;
    ObsMinAngle : ARRAY [1..MaxSquadron] OF REAL;
    ObsMinContrast : ARRAY [1..MaxSquadron] OF REAL;
    ObsXpire : INTEGER;
    ProcDelay : REAL;
    Priority : INTEGER;
    RecRate : REAL;
    Report : INTEGER;
    SendRate : REAL;
    SquadIntensity : ARRAY [1..MaxSquadron] OF INTEGER;
    SquadLength : REAL;
    Squadrons : INTEGER;
    SquadSep, RowSep : REAL;
    SquadWidth : REAL;
    Theta : REAL;
    VO : ARRAY [1..MaxSquadron] OF REAL;
    VWEffect : REAL;
    Weapon : ARRAY [1..MaxWType] OF WRec;
    WeapPriority : ARRAY [1..MaxBattalion] OF ARRAY [1..MaxWType] OF INTEGER;
    WeapSensitivity : ARRAY [1..MaxWType] OF REAL;
    Wear : ARRAY [1..MaxSquadron] OF REAL;
    X, Y : REAL;
END;

ComMsg      = RECORD
    Army : BOOLEAN;
    Dest : INTEGER;
    Time : INTEGER;
    msg : Battalion;
    Priority : INTEGER;
END;

```

## VARIABLES

```
VAR  Army : Array [BOOLEAN] OF Array [1..MaxBattalion] OF Battalion;  
     NArmy : Array [BOOLEAN] OF INTEGER;  
     Terrain : Array [0..MaxTerrain, 0..MaxTerrain] OF INTEGER;  
     Weather : Array [1..MaxWeather] of WEvent;  
     Cmsgs : Array [BOOLEAN] OF Array [1..MaxMsg] OF ComMsg;  
     NCmsgs : Array [BOOLEAN] OF INTEGER;  
     Duration : INTEGER;  
     Params : PRec;  
     Location : Array [BOOLEAN] OF Array [1..MaxBattalion] OF  
               RECORD  
                 X, Y : REAL;  
                 W, H : REAL;  
               END;  
  
     Status : Array [BOOLEAN] OF Array [1..MaxBattalion] OF  
             RECORD  
               Functional : INTEGER;  
               Casualties : INTEGER;  
               Destroyed : INTEGER;  
             END;  
  
     Action : Array [BOOLEAN] OF  
             RECORD  
               Array [1..MaxBattalion, 1..MaxBattalion] OF  
               Results;
```

## APPENDIX B

The following is an extract of source code from test program six. The original source code lines are numbered. The unnumbered lines within the starred lines are annotated with the appropriate fault number, the fault description, and one possible solution.

```
procedure InitBattalion

(IArmy : boolean;
 IBatt : integer;
 Params : PRec);

var Squad, Squad1, WeaponType, Weapon : integer;

    LocatIntensity : real;

001 begin                {InitBattalion}
002   with Army[IArmy, IBatt] do
003     AlignSquads(IArmy, IBatt, Squadrons, X, Y, OldArmy
        [IArmy, IBatt].SquadArray);
004   Sqd1 := 1;
005   for Squad := 1 to Army[IArmy, IBatt].Squadrons do
006     with OldArmy[IArmy, IBatt].SquadArray[Sqd1] do

007       begin
008         ArmyIndex := Squad;
009         dKilled := 0;
010         Fixing := 0;
011         if Army[IArmy, IBatt].Endurance[Squad] < 0 then
012           TellError ('Negative endurance when initializing battalion')
013         else
014           if Army[IArmy, IBatt].Endurance[Squad] > 0 then

015             begin
016               Endurance := Army[IArmy, Batt].Endurance[Sqd];
017               Sqd1 := Sqd1 + 1;
018             end;

019           Status := Functional;
020         end;

021   for WeaponType := 1 to Params.NumWTypes do
022     with OldArmy[IArmy, IBatt].Weapons[WeaponType] do
```

```

023     begin
024         if Army[IArmy,IBatt].Weapon[WeaponType].NumWeapon < 0 then
025             TellError('Negative NumWeapons when initializing bn')
026         else

027             begin
028                 NumWeapons := Army[IArmy, IBatt].Weapon
                    [WeaponType].NumWeapon;
029                 NumAvail := NumWeapons;
030             end;

```

\*\*\*\*\*

Fault 1 : Undefined pointer reference due to no initialization of AttList

One possible fix :

AttList := nil;

\*\*\*\*\*

```

031         dNumWeapons := 0;
032         for Weapon := 1 to MaxWeapon do
033             Uses[Weapon] := 0;
034         end;

035     with Army[IArmy, IBatt] do

036         begin
037             OldArmy[IArmy, IBatt].NumNewCasualties := 0;
039             OldArmy[IArmy, IBatt].NumCasualties := 0;
040             OldArmy[IArmy, IBatt].NumMsgsProcessing := 0;
041             OldArmy[IArmy, IBatt].dNumRestored := 0;
042             OldArmy[IArmy, IBatt].dSquadrons := 0;
043             OldArmy[IArmy, IBatt].Observations.SomethingSeen := false;
044             if (X < 0) or (X > (Params.XDelta*MaxTerrain)) then
045                 TellError('Battalion X is out of range during Initialize')
046             else
047                 OldArmy[IArmy, IBatt].X := X;
048             if (Y < 0) or (Y > (Params.YDelta*MaxTerrain)) then
049                 TellError('Battalion Y is out of range during Initialize')
050             else
051                 OldArmy[IArmy, IBatt].Y := Y;
052             if (Squadrons > 0) or (Squadrons > MaxSquadron) then

```

\*\*\*\*\*

Fault 2 - Destroyed battalions not initialized.

One possible fix :

```
begin
  TellError('Squadrons is negative or too large in initialize');
  if (Squadrons < 0) then
    OldArmy[IArmy, IBatt].Squadrons := 0
  else
    OldArmy[IArmy, IBatt].Squadrons := MaxSquadron;
end

else
  TellError('Squadrons is negative or too large in initialize');
```

\*\*\*\*\*

```
053   else
054     OldArmy[IArmy, IBatt].Squadrons := Sqd1 - 1 {Squadrons};
```

\*\*\*\*\*

Fault 3 - Report refers to nonexistent battalions - message superfluous.

One possible fix :

```
for Sqd := 1 to NArmy[notIArmy] do
  OldArmy[IArmy, IBatt].Observations.
  NumObserved[Sqd1] := 0;
```

\*\*\*\*\*

```
055   if (NumJammers < 0) or (NumJammers > Squadrons) then
056     TellError('NumJammers in impossible range in initialize')
057   else
058     OldArmy[IArmy, IBatt].NumJammers := NumJammers;
059   if (NumFixers < 0) or (NumFixers > Squadrons) then
060     TellError('NumFixers in impossible range in initialize')
061   else
062     OldArmy[IArmy, IBatt].NumFixers := NumFixers;
063   if (NumProcess < 0) or (NumProcess > Squadrons) then
064     TellError('NumProcess in impossible range in initialize')
065   else
```



```

066      OldArmy[IArmy, IBatt].NumProcess := NumProcess;
067      if (NumReceive < 0) or (NumReceive > Squadrons) then
068        TellError('NumReceive in impossible range in initialize')
069      else
070        OldArmy[IArmy, IBatt].NumReceive := NumReceive;
071        if (NumSend < 0) or (NumSend > Squadrons) then
072          TellError('NumSend in impossible range in initialize')
073        else
074          OldArmy[IArmy, IBatt].NumSend := NumSend;
075          if FixSuppl < 0 then
076            TellError('FixSuppl negative during initialize')
077          else
078            OldArmy[IArmy, IBatt].FixSuppl := FixSuppl;
079            UpdateBattalionVelocity(IArmy, IBatt);
080            OldArmy[IArmy, IBatt].Velocity := NewArmy[IArmy, IBatt].Velocity;
081      end; {with}

```

```

082 end;          {InitBattalion}

```

```

procedure Initialize

```

```

  (Params :PRec;
   Duration : integer;
   var Time : integer);

```

```

  var Side : boolean;
  Batt, WEvent, EnemyBatt : integer;

```

```

procedure CreateLOSList

```

```

  var I : integer;
  NewRecord : CoordPtr;

```

```

083 begin          {CreateLOSList}
084   LOSLBase := nil;
085   for I := 1 to (Params.Sample Rate - 2) do

086     begin
087       new(NewRecord);
088       NewRecord^.Next := LOSLBase;
089       LOSLBase := NewRecord;
090     end;
091 end;          {CreateLOSList}

```

```

092 begin           {Initialize}
093   if Duration <= 0 then
094     TellError ('Duration is negative or zero');
095   Time := 0;
096   CreateLOSList;
097   with Params do

098     begin
099       if (NumWTypes < 0) or (NumWTypes > MaxWType) then
100         TellError ('NumWTypes is out of possible range');
101       if XDelta <= 0 then
102         TellError ('XDelta is not greater than zero');
103       if YDelta <= 0 then
104         TellError ('YDelta is not greater than zero');
105       if (NumWEvents < 0) or (NumWEvents > MaxWeather) then
106         TellError ('NumWEvents is out of possible range');
107       if SampleRate < 0 then
108         TellError ('SampleRate is negative');
109       if IMeanAlt <= 0 then
110         TellError ('IMeanAlt is not positive');
111     end;

112   for WEvent := 1 to Params.NumWEvents do
113     with Weather[WEvent] do

114       begin
115         if TStart > TEnd then
116           TellError ('Bad starting and ending time for weather event');
117         if WRadius <= 0 then
118           TellError ('Bad radius for weather event');

119         if WSeverity > Params.WMaxSeverity then
120           TellError ('WEvent Severity > Params.WMaxSeverity');
121       end;

122   for Side := false to true do
123     for Batt := 1 to NArmy[Side] do

124       begin
125         CheckBattConstants(Army[Side, Batt], Params.NumWTypes);
126         InitBattalion(Side, Batt, Params);
127         for EnemyBatt := 1 to NArmy[not Side] do
128           Action[Side, Batt, EnemyBatt] := None;
129       end;

130   NewArmy := OldArmy;
131 end;           {Initialize}

```

```
132 begin          {CONFLICT}
133   Initialize (Params, Duration, Time);
134   PerformSimulation (Params, Duration, Time);
135   DetermineOutput;
136 end;            {CONFLICT}
```

## APPENDIX C

The following is an extract of source code from test program six. The original source code lines are numbered. The unnumbered lines within the starred lines are annotated with the appropriate fault number, the fault description, and one possible solution.

```
procedure PutInCommands

(CommandsFinished : CommandPtr);

var TempPointer : CommandPtr;
    Observations : BattalionObservations;
    Targets : WeaponArray;
    NumberMsgsProcessing, WeaponType : integer;

001 begin          {PutInCommands}
002   while CommandsFinished <> nil do
003     with CommandsFinished^ do

004       begin

*****

Fault 1 - Command messages implemented in destroyed battalions.

One possible fix :

        if OldArmy[DestArmy, Dest].Squadrons > 0 then

            begin

*****

005           writeln (Time : 2,': Including command sent to
                    'DestArmy,',',Dest : 1);
```

\*\*\*\*\*

Fault 2 - Incorrect variable used in procedure call and assignment statement.

One possible fix :

```
CheckBattalionConstants (Cmsgs[DestArmy, msg].msg,  
    Params.NumWTypes);  
Army[DestArmy, Dest] := Cmsgs [DestArmy, msg].msg;
```

\*\*\*\*\*

```
006      CheckBattalionConstants (msg, Params.NumWTypes);  
007      Army[DestArmy, Dest] := msg;  
008      Observations := OldArmy[DestArmy, Dest].Observations;  
009      NumberMsgsProcessing := OldArmy[DestArmy, Dest].NumMsgsProcessing;  
010      Targets := OldArmy[DestArmy, Dest, Weapons];  
011      InitBattalion (DestArmy, Dest, Params);  
012      OldArmy[DestArmy, Dest].Observations := Observations;  
013      OldArmy[DestArmy, Dest].NumMsgsProcessing := NumMsgsProcessing;  
014      for WeaponType := 1 to Params.NumWTypes do  
015      OldArmy[DestArmy, Dest].Weapons[WeaponType].AttList := Targets[WeaponType].  
      AttList;
```

\*\*\*\*\*

Fault 3 - NewArmy not updated to match OldArmy after command messages.

One possible fix :

```
NewArmy[DestArmy, Dest] := Old Army[DestArmy, Dest];
```

\*\*\*\*\*

```
016      end;  
017      TempPointer := next;  
018      dispose (CommandsFinished);  
019      CommandsFinished := TempPointer;  
020      end;
```

```
021 end;      {PutInCommands}
```

## APPENDIX D

The following is an extract of source code from test program three. The original source code lines are numbered. The unnumbered lines within the starred lines are annotated with the appropriate fault number, the fault description, and one possible solution.

```
procedure Restore

(Armynun : boolean;
 Batt : integer);

var Killed, NewKilled, Sqd : integer;
    TotRestoration : real;

001 begin          {Restore}
002   with ArmyTemp[Armynun][Batt] do

003     begin
004       TotRestoration := 0;
005       if (Army[Armynun][Batt].Squadrons - Killed >= 1 then
006         for sqd := 1 to Army[ArmyNum][Batt].Squadrons do

007           begin

*****

Fault 1 - Restoration allowed when FixSuppl, FixRate, and NumFixers < 0.

One possible fix :

           if SquadStat[Sqd] and (FixSuppl > 0) and (FixRate > 0) and (NumFixers > 0) then

*****

008           if SquadStat[Sqd] then
009             Restoration[Sqd, 0] := min (min (FixRate *
              NumFixers/Casualties, FixSuppl/Casualties),
              Endurance0[Sqd] - Endurance[Sqd])
010           else
011             Restoration[Sqd, 0] := 0;
012             TotRestoration := TotRestoration + Restoration[Sqd, 0];
013           end;
```

```
014      dFixSuppl := -min(TotRestoration, Army[ArmyNum][Batt].FixRate *  
      Army[ArmyNum][Batt].NumFixers);
```

```
015      end;
```

```
016 end;      {Restore}
```

## APPENDIX E

The following is an extract of source code from test program three. The original source code lines are numbered. The unnumbered lines within the starred lines are annotated with the appropriate fault number, the fault description, and one possible solution.

```
procedure InitVals

var ArmyNum : boolean;
    Batt, EBatt, Sqd, WeapType, tim : integer;

001 begin                {InitVals}
002   HeadQueue := nil;
003   for ArmyNum := false to true do
004     for Batt := 1 to NArmy[ArmyNum] do

005       begin
006         for Sqd := 1 to MaxBattalion do
007           for EBatt := to MaxBattalion do

008             begin
009               ObsLocList[ArmyNum, Sqd, EBatt].NumList := 0;
010               ObsLocList[ArmyNum, Sqd, EBatt].Head := nil;
011             end;
012             with ArmyTemp[ArmyNum][Batt] do

013               begin
014                 Casualties := 0;
015                 dFixSuppl := 0;
016                 dFunctional[0] := 0;
017                 dFunctional[1] := 0;
018                 dJam := 0;
019                 for tim := 0 to 1 do

020                   begin
021                     dNumFixers[tim] := 0;
022                     dNumJammers[tim] := 0;
023                     dNumProcess[tim] := 0;
024                     dNumReceive[tim] := 0;
025                     dNumSend[tim] := 0;
026                   end;
027                   dX := 0;
028                   dY := 0;
029                   Velocity := 9999;
```



```

030      Killed := 0;
031      for Sqd := 1 to Army[ArmyNum][Batt].Squadrons do

032          begin

```

\*\*\*\*\*

Fault 1 - Initial velocity counted incorrect if Endurance <= 0.

One possible solution :

```

      if Army[ArmyNum][Batt].Endurance[Sqd] > 0 then

```

\*\*\*\*\*

```

033          Velocity := min(Velocity,
                           Army[ArmyNum][Batt].VO[Sqd];
034          SquadStat[Sqd] := false;
035          dEndurance[Sqd] := 0;
036          for tim := 0 to 1 do

037              begin
038                  Restoration[Sqd, tim] := 0;
039              end;

040          Endurance0[Sqd] := Army[ArmyNum][Batt].Endurance[Sqd];
041          end;
042          Functional := Army[[ArmyNum][Batt].Squadrons;
043          for WeapType := 1 to Params.NumWTypes do

044              begin
045                  for EBatt := 1 to NArmy[not ArmyNum] do
046                      NumWeapUsedOn[EBatt], WeapType] := 0;
047                      KillersAvail[WeapType] := Army[ArmyNum]
                        [Batt].Weapon[WeapType].NumWeapon;
048              end;

049          NewHurt[0] := 0;
050          NewHurt[1] := 0;
051          NumJammed := 0;
052          for EBatt := 1 to NArmy[not ArmyNum] do

053              begin
054                  Action[ArmyNum][Batt, EBatt] := None;
055              end;

```

```

056         end;
057     end;
058 end;          {InitVals}

059 begin          {Conflict}
060     InitVals;
061     if Duration > 0 then
062         for Mainloop := 0 to Duration do

063             begin
064                 DoAction (Mainloop);
065                 UpDate (Mainloop);
066                 Output (Mainloop);
067             end

068         else
069             TellError ('Invalid Duration value');
070 end;          {Conflict}

```

## APPENDIX F

This appendix is divided into two sections and contains the detailed definition of the software failure regions analyzed during the course of this research. Section A addresses those failure regions that address only the internal conditions. Section B includes those complete failure regions that include both the internal and external conditions. The failure regions are numbered in accordance with the fault numbers originally identified during testing of the source code. Additionally, a brief description of the fault is included.

### A. FAILURE REGIONS WITH INTERNAL CONDITIONS

**Fault 3.1 :** Improper count on number of busy processors in procedure ReceiveMessages.

Condition I :  $(PTR \neq NIL) \text{ AND } ((PTR^{DestArmy} = B) \text{ AND } (PTR^{DestBatt} = f))$

Condition II :  $((PTR^{Processed} = recd) \text{ AND } (PTR^{TimeRecd} = MainLoop))$

Condition III : TRUE

**Fault 3.2 :** Violation of queue structure when messages of equal priority are ready for processing.

Condition I : TRUE

Condition II : TRUE

Condition III : TRUE

**Fault 3.3 :** Messages implemented by destroyed battalion in procedure FollowCommandMessages.

Condition I :  $Match \neq NIL$

Condition II :  $Army[ArmyIndex, BattIndex].Squadrons - ArmyTemp[ArmyIndex, BattIndex].Killed \leq 0$

Condition III : TRUE

**Fault 3.4 :** Initial Endurance  $\leq 0$  reported erroneous in command messages.

Condition I : (Match  $\neq$  NIL)  
AND (Army[ArmyIndex, BattIndex].Squadrons - ArmyTemp[ArmyIndex, BattIndex].Killed  $> 0$ )  
AND ((i  $\in$  [1..Squadrons]) AND (Squadrons  $> 0$ ))

Condition II : (  $\exists$  (e) | Endurance[i] (e)  $< 0$ )

Condition III : (  $\exists$  (s) | (s  $> i$ ) AND (s  $\leq$  Squadrons) AND (Endurance[s]  $<$  Endurance [i])

**Fault 3.9 :** Incorrect variables used in calculation of damage.

Condition I : (Army[ArmyNum][Batt].Squadrons - ArmyTemp[ArmyNum][Batt].Killed  $> 0$ )  
AND ((EBatt  $\in$  [1..NArmy[EArmy]]) AND (NArmy[EArmy]  $> 0$ ))  
AND ((WeapType  $\in$  [1..Params.NumWTypes]) AND (Params.NumWTypes  $> 0$ ))  
AND (ArmyTemp[ArmyNum][Batt].NumWeaponsUsedOn[Batt][WeapType]  $> 0$ )  
AND ((WeapToUse  $\in$  [1..ArmyTemp[ArmyNum[Batt].NumWeaponsUsedOn  
[EBatt][WType]) AND (ArmyTemp[ArmyNum[Batt].NumWeaponsUsedOn  
[EBatt][WType]  $> 0$ ))  
AND ((ESquad  $\in$  [1..Army[EArmy][EBatt].Squadrons)  
AND (Army[EArmy][EBatt].Squadrons  $> 0$ ))

Condition II : TRUE

Condition III : TRUE

**Fault 3.11 :** Incorrect check in comparing variable M against MaxTerrain.

Condition I : TRUE

Condition II : (  $\exists$  (x) | M(x) = MaxTerrain)

Condition III : TRUE

**Fault 3.12 :** Variable TM undefined when Dist = 0.

Condition I : (dist  $\leq 0$ )

Condition II : (dist  $\leq 0$ )

Condition III : TRUE

**Fault 3.13 :** Observation from side uses wrong points in angle calculation.

Condition I : TRUE

Condition II : ((SquadLoc[EnemyArmy][e][k].x  $\diamond$  TR.x) OR ((SquadLoc[EnemyArmy][e][k].y  $\diamond$  TR.y)  
AND (SquadLoc[EnemyArmy][e][k].y  $\diamond$  BR.y)))  
OR ((SquadLoc[EnemyArmy][e][k].x  $\diamond$  TL.x) OR ((SquadLoc[EnemyArmy][e][k].y  
 $\diamond$  TL.y) AND (SquadLoc[EnemyArmy][e][k].y  $\diamond$  BL.y)))  
OR ((SquadLoc[EnemyArmy][e][k].x  $\diamond$  TR.x) AND (SquadLoc[EnemyArmy][e][k].x  
 $\diamond$  TL.x))

Condition III : ((Angle > Army[B][i].ObsMinAngle[j]) AND (Params.SampleRate >= 2))

**Fault 3.16 :** Segmentation violation in Procedure FindA when battalion leaves undefined terrain.

Condition I : TRUE

Condition II :  $\exists (x) \mid ((M(x) \geq \text{MaxTerrain}) \text{ OR } (N(x) \geq \text{MaxTerrain}))$

Condition III : TRUE

**Fault 3.18 :** Variable N undefined in Procedure WeaponsSighting at first occurrence.

Condition I : TRUE

Condition II : TRUE

Condition III : TRUE

**Fault 3.19 :** Variable M undefined in Procedure WeaponsSighting at first occurrence.

Condition I : TRUE

Condition II : TRUE

Condition III : TRUE

**Fault 3.20 :** Incorrect check in comparing variable N against MaxTerrain.

Condition I : TRUE

Condition II : (  $\exists (x) \mid N(x) = \text{MaxTerrain}$  )

Condition III : TRUE

**Fault 3.22 :** Restoration allowed when FS < 0.

Condition I : (Army[ArmyNum][Batt].Squadrons - Killed >= 1)  
AND ((Sqd  $\in$  [1..Army[ArmyNum][Batt].Squadrons)  
AND (Army[ArmyNum][Batt].Squadrons > 0))

Condition II : ((FixSuppl <= 0) OR (FixRate <= 0) OR NumFixers <= 0))

Condition III : (  $\exists (s) \mid (s > \text{Sqd}) \text{ AND } (s \leq \text{Army[ArmyNum][Batt].Squadrons})$  )  
AND [min (min (FixRate \* NumFixers/Casualties, FixSuppl/Casualties),  
Endurance[s] > Restoration[s, 0))

**Fault 3.23 :** Messages lost if NumProcess goes transiently to 0.

Condition I : ((B  $\in$  [false]  $\mid$  NArmy[false] > 0) AND (f  $\in$  [1..NArmy[false]]))  
AND ((B  $\in$  [true]  $\mid$  NArmy[true] > 0) AND (f  $\in$  [1..NArmy[true]]))  
AND (rec  $\neq$  NIL)  
AND (Army[B][f].NumProcess <= ProcBusy)

Condition II : (Army[B][f].NumProcess <= 0)

Condition III : TRUE

**Fault 3.24 :** Positions not initialized when Duration = 0.

Condition I : ((ArmyNum  $\in$  [true]  $\mid$  NArmy[true] > 0) AND (Batt  $\in$  [1..NArmy[true]]))  
AND ((ArmyNum  $\in$  [false]  $\mid$  NArmy[false] > 0) AND (Batt  $\in$  [1..NArmy[false]]))  
AND ((Sqd  $\in$  [1..Army[ArmyNum][Batt].Squadrons)  
AND (Army[ArmyNum][Batt].Squadrons > 0))

Condition II : TRUE

Condition III : TRUE

**Fault 3.26 :** Initially destroyed battalions not considered in calculation 'Killed.'

Condition I : ((ArmyNum  $\in$  [true]  $\mid$  NArmy[true] > 0) AND (Batt  $\in$  [1..NArmy[true]]))  
AND ((ArmyNum  $\in$  [false]  $\mid$  NArmy[false] > 0) AND (Batt  $\in$  [1..NArmy[false]]))  
AND ((Sqd  $\in$  [1..Army[ArmyNum][Batt].Squadrons)  
AND (Army[ArmyNum][Batt].Squadrons > 0))

Condition II : (Army[ArmyNum][Batt].Endurance[Sqd] <= 0)

Condition III : TRUE

**Fault 3.28 :** Procedure FindA returns incorrectly defined value if  $X < 0$ .

Condition I : TRUE

Condition II : ( $\exists (x) \mid ((M(x) < 0) \text{ OR } (N(x) < 0))$ )

Condition III : TRUE

**Fault 3.43 :** Improper observation allowed when  $SR < 2$ .

Condition I : (Army[not B][e].Endurance[k] > 0)  
AND ((k  $\in$  [1..Army[EnemyArmy][e].Squadrons])  
AND (Army[EnemyArmy][e].Squadrons > 0))  
AND ((e  $\in$  [1..NArmy[EnemyArmy]]) AND (NArmy[EnemyArmy] > 0))  
AND (Army[B][i].Endurance[j] > 0)  
AND ((j  $\in$  [1..Army[B][i].Squadrons]) AND (Army[B][i].Squadrons > 0))  
AND ((B  $\in$  [true]  $\mid$  NArmy[true] > 0) AND (i  $\in$  [1..NArmy[true]]))  
AND ((B  $\in$  [false]  $\mid$  NArmy[false] > 0) AND (i  $\in$  [1..NArmy[false]]))

Condition II : ((Params.SampleRate < 2) AND (Angle > Army[B][i].ObsMinAngle[j]))

Condition III : ((Angle > Army[B][i].ObsMinAngle[j]) AND (Params.SampleRate > 2))

**Fault 3.44 :** Improper variable assignment.

Condition I : ((EBatt  $\in$  [1..NArmy[EArmy]] AND (NArmy[EArmy] > 0))  
AND ((WearType  $\in$  [1..Params.NumWTypes]) AND (Params.NumWTypes > 0))

Condition II : (Army[ArmyNum][Batt].Weapon[WeapType].FireRate <= 0)

Condition III : TRUE

**Fault 3.45 :** Variables improperly set in destroyed battalions.

Condition I : TRUE

Condition II : (Army[ArmyNum][Batt].Squadrons - ArmyTemp[ArmyNum][Batt].Killed <= 0)

Condition III : TRUE

**Fault 6.1 :** Reversed parameters leads to improper observation results.

Condition I : (AngleSubGreater = TRUE)

Condition II : ((SXe <> SXg) OR (SYe <> SYg))

Condition III : TRUE

**Fault 6.2 :** No check on send time if commands of equal priority are implemented at same time.

Condition I : (CommandsFinished <> NIL) AND OldArmy[DestArmy, Dest].Squadrons > 0)

Condition II : (Cmsgs[DestArmy, msg].msg <> CommandsFinished^.msg

Condition III : TRUE

**Fault 6.3 :** Improper calculation of variable due to incorrect denominator.

Condition I : TRUE

Condition II : TRUE

Condition III : (((SXg <> SXe) OR (SYg <> SYe))

**Fault 6.5 :** Incorrect calculation - change in available weapons used vice running sum.

Condition I : ((IWeap ∈ [1..MaxWType] AND (MaxWType > 0))

Condition II : (OldArmy[IArmy, IBatt].Weapons[IWeap].NumAvail > 0)

Condition III : TRUE



**Fault 6.6 :** Divide by zero when NumWeapons = 0 when assigning target coordinates.

Condition I : ((EBatt ∈ [1..NArmy[not(IArmy)]]) AND (NArmy[not(IArmy)] > 0))  
AND (OldArmy[not(IArmy), EBatt].Squadrons > 0)  
AND (LWeap ∈ [1..Params.NumWTypes]) AND (Params.NumWTypes > 0))  
AND ((I ∈ [1..NumWeapons(IArmy, IBatt, EBatt, LWeap)])  
AND (NumWeapons(IArmy, IBatt, EBatt, LWeap) > 0))

Condition II : (OldArmy[IArmy, IBatt].Weapons[LWeap].NumWeapons = 0)

Condition III : TRUE

**Fault 6.7 :** Undefined pointer reference due to no re-initialization of AttList.

Condition I : TRUE

Condition II : TRUE

Condition III : TRUE

**Fault 6.8 :** Undefined pointer reference due to no initialization of AttList.

Condition I : ((WeaponType ∈ [1..Params.NumWtypes]) AND (Params.NumWTypes > 0))

Condition II : TRUE

Condition III : TRUE

**Fault 6.10 :** Report refers to nonexistent battalion - message superfluous.

Condition I : TRUE

Condition II : TRUE

Condition III : TRUE

**Fault 6.11 :** Message improperly deleted if more recently sent message is placed ahead of it.

**Condition I :** (CommandsFinished  $\neq$  NIL) AND (WhereFinished  $\neq$  NIL)  
AND [(WhereInFinished^.Priority  $\neq$  CommandToCollect.Priority  
AND ((WhereInFinished^.Priority  $\neq$  CommandToCollect^.Priority)  
OR (CMsgs[WhereInFinished^.DestArmy, WhereInFinished^.msg].Time <  
CMsgs[CommandToCollect^.msg].Time))]

**Condition II :** TRUE

**Condition III :** TRUE

**Fault 6.12 :** Improper variable used to calculate jamming.

**Condition I :** (OldArmy[DestArmy, DestBatt].NumReceive > 0)  
AND ((Batt  $\in$  [1..NArmy[not DestArmy]) AND (NArmy[not DestArmy] > 0))  
AND (Army[not DestArmy, Batt].CommJamEff > 0)

**Condition II :** Army[not DestArmy, Batt].CommJamRadius  $\neq$  Army[not DestArmy, Batt].CommJamEff  
AND (Army[not DestArmy, Batt].CommJamRadius - Distance(OldArmy  
[not DestArmy, Batt].X, OldArmy[not DestArmy, Batt].Y,  
OldArmy[DestArmy, Batt].X, OldArmy[DestArmy, Batt].Y)

**Condition III :** TRUE

**Fault 6.13 :** Command messages implemented in destroyed battalions.

**Condition I :** (CommandsFinished  $\neq$  NIL)

**Condition II :** (OldArmy[DestArmy, Dest].Squadrons > 0)

**Condition III :** TRUE

**Fault 6.14 :** Variable InOwnObs not initialized.

Condition I : (ReportsFinished  $\neq$  NIL)  
AND ((EnemyBatt  $\in$  [1..NArmy[not DestArmy]]) AND (NArmy[not DestArmy] > 0))  
AND (OldArmy[not DestArmy, EnemyBatt].Squadrons > 0)  
AND ((SentSquadObs  $\in$  [1..SumofSentObsToOneBn.NumObserved[EnemyBatt])  
AND (SumofSentObsToOneBn.NumObserved[EnemyBatt] > 0))

Condition II : TRUE

Condition III : TRUE

**Fault 6.15 :** NewArmy not updated to match OldArmy after command messages.

Condition I : ((CommandsFinished  $\neq$  NIL) AND (OldArmy[DestArmy, Dest].Squadrons > 0))

Condition II : TRUE

Condition III : TRUE

**Fault 6.16 :** Observation improperly implemented.

Condition I : TRUE

Condition II : (P = NIL) AND (Params.SampleRate  $\neq$  2)

Condition III : TRUE

**Fault 6.17 :** NumWeapons > 0 when Army.Weapon.FireRate = 0.

Condition I : TRUE

Condition II : (OldArmy[IAmy, IBatt].Weapon[IWeap].FireRate  $\leq$  0)

Condition III : (OldArmy[IAmy, IBatt].Weapons[IWeap].NumAvail  $\neq$  0)  
OR ((KP < LL) AND (KP > KAKF)) OR ((KP  $\geq$  LL) AND (LL > KAKF))

**Fault 6.18 :** Report messages blocked by bad test in procedure IncludeCommObs.

Condition I : (NextReport  $\neq$  NIL) AND (NextReport^.Finished = Time)

Condition II : (OldArmy[DestArmy, Dest].Squadrons > 0)  
AND ((Time - Army[DestArmy, Dest].ObsXpire) > TimeSent)

Condition III : TRUE

**Fault 6.20 :** InitBattalion doesn't initialize destroyed battalions.

Condition I : TRUE

Condition II : (Army[IArmy, IBatt].Squadrons < 0)  
OR (Army[IArmy, IBatt].Squadrons > MaxSquadron)

Condition III : TRUE

## B. FAILURE REGIONS WITH INTERNAL AND EXTERNAL CONDITIONS

**Fault 3.10 :** Duration = 0 is considered erroneous when it's not.

Condition I : (Duration <= 0)

Condition II : (Duration = 0)

Condition III : TRUE

**Fault 3.17 :** Result variables not set if Duration < 0.

Condition I : (Duration <= 0)

Condition II : (Duration < 0)

Condition III : TRUE

**Fault 3.5 :** Initial velocity counted incorrect if Endurance <= 0.

Condition I : ((ArmyNum ∈ [true] | NArmy[true] > 0) AND (Batt ∈ [1..NArmy[true]]))  
AND ((ArmyNum ∈ [false] | NArmy[false] > 0) AND (Batt ∈ [1..NArmy[false]]))  
AND ((Sqd ∈ [1..Army[ArmyNum][Batt].Squadrons])  
AND (Army[ArmyNum][Batt].Squadrons > 0))

Condition II : (Army[ArmyNum][Batt].Endurance[Sqd] <= 0)  
AND ((Army[ArmyNum][Batt].VO[Sqd] < y  
where ((y ∈ [x | x = 9999]) OR ( ∃ (s) | 1 <= s <= (Sqd - 1)  
AND x = Army[ArmyNum][Batt].VO[s]))

Condition III : ((( ∃ (s) | (s > Sqd)) AND (s <= Army[ArmyNum][Batt].Squadrons))  
AND (Army[ArmyNum][Batt].VO[s] < Velocity))

**Fault 6.4 :** Incorrect equality check in repeat...until loop.

Condition I : TRUE

Condition II : (Duration <= 0)

Condition III : TRUE

## LIST OF REFERENCES

Alberts, D. S. "The Economics of Software Quality Assurance Proceedings of the 1976 National Computer Conference. New York, NY, AFIPS Press, 1976, pp. 230 - 238.

Amman, P.E. and Knight, J.C. "Data Diversity : An Approach to Software Fault Tolerance." **IEEE Transactions on Computers**. April 1988, pp. 418 - 425.

Bahr, Jeffrey L. "A Study of the Factors Affecting Software Testing Performance and Computer Program Reliability Growth." PhD Dissertation, University of Southern California, June 1980.

Basili, V.R. and Selby, R.W. "Comparing the Effectiveness of Software Testing Strategies." **IEEE Transactions on Software Engineering**, Vol. SE - 13, No. 12, December 1987, pp. 1278 - 1296.

Beizer, Boris. **Software Testing Techniques**. Van Nostrand Reinhold, New York, NY, 1983.

Blum, B.I. "On Data and Their Analysis." **ACM SIGSOFT SOFTWARE ENGINEERING NOTES**. Vol 14, No. 5, July 1989, pp. 24 - 34.

Cavano, Joseph P. "Toward High Confidence Software." **IEEE Transactions on Software Engineering**. Vol SE-11, No. 12, December 1985.

DeMillo, Richard A., Lipton, Richard J. and Sayward, Frederick G. "Hints on Test Data Selection: Help for the Practicing Programmer." **Computer**. April 1978, pp. 34 - 41.

"Glossary of Software Engineering Terminology." **ANSI-IEEE Std 729-1983**, Institute of Electrical and Electronics Engineers, 1983.

Hetzel, W.C. "An Experimental Analysis of Program Verification Methods." Ph.D. Dissertation, University of North Carolina at Chapel Hill, Chapel Hill, NC, 1976.

Lamb, David Alex. **Software Engineering : Planning for Change**. Prentice Hall, Englewood Cliffs, NJ, 1988.

Myers, Glenford J. **The Art Of Software Testing**. John Wiley and Sons, New York, NY, 1979.

Neumann, Peter G. "RISKS : Cumulative Index of Software Engineering Notes." Vol 14, No. 1, January 1989, pp. 22 - 26.

Reifer, D. J. "The Software Engineering Checklist." **Proceedings, Computers in Aerospace Conference.** 1977, pp. 125 - 129.

Shelly, Gary B. and Cashman, Thomas J. **COMPUTER FUNDAMENTALS for an Information Age.** Anaheim Publishing Company, Brea, CA, 1984.

Shimeall, Timothy. "FALTER - A Fault Annotation Tool." Technical Report NPS52-89-051, Naval Postgraduate School, Monterey, CA, September, 1989.

Shimeall, Timothy. "REACHER - A Reachability Condition Derivation Tool." Technical Report NPS52-89-050, Naval Postgraduate School, Monterey, CA, September, 1989.

Shimeall, Timothy. "An Empirical Comparison Comparison of Software Fault Tolerance and Fault Elimination." University of California, Irvine, Irvine, CA, 1989.

Shimeall, Timothy and Griffin, Rachel. "A Tool Set for the Analysis of Software Failure Regions." Naval Postgraduate School, Monterey, CA, Submitted for publication, 1989.

Shimeall, Timothy and Leveson, Nancy. "An Empirical Comparison of Software Fault Tolerance and Fault Elimination." Technical Report NPS52-89-047, Naval Postgraduate School, Monterey, CA, July 1989.

Voas, Jeffrey M. and Morell, Larry J. "Fault Sensitivity Analysis (PIA) Applied to Computer Programs." Technical Report WM-89-4, College of William and Mary, Williamsburg, VA, 10 December 1989.

## **BIBLIOGRAPHY**

Aldenderfer, Mark S. and Blashfield, Roger K. Series: Quantitative Applications in the Social Sciences. **CLUSTER ANALYSIS**. SAGE Publications, Beverly Hills, CA, 1986.

Adrian, W. Richards, Branstad, Martha A., and Cherniavsky, John C. "Validation, Verification, and Testing of Computer Software." **ACM Computing Surveys**. June 1982, pp. 159 - 192.

Alberts, D. S. "The Economics of Software Quality Assurance." **Proceedings of the 1976 National Computer Conference**. New York, NY, AFIPS Press, 1976, pp. 230 - 238.

Ammann, Paul E. "An Approach to Software Fault Tolerance." PhD Dissertation, University of Virginia, Charlottesville, VA, 1988.

Ammann, P.E. and Knight, J.C. "Data Diversity : An Approach to Software Fault Tolerance." **IEEE Transactions on Computers**. April 1988, pp. 418 - 425.

Avizienis, Algirdas. "The N-Version Approach to Fault Tolerant Software." **IEEE Transactions on Software Engineering**. Vol SE - 11, No. 12, December 1985, pp. 1491 - 1501.

Bahr, Jeffrey L. "A Study of the Factors Affecting Software Testing Performance and Computer Program Reliability Growth." PhD Dissertation, University of Southern California, June 1980.

Barbour, Ahmed E. and Wojcik, Anthony S. A General, "Constructive Approach to Fault-Tolerant Design Using Redundancy." **IEEE Transactions on Computers**. Vol 38, No. 1, January 1989, pp. 15 - 26.

Basili, V.R. and Selby, R.W. "Comparing the Effectiveness of Software Testing Strategies." **IEEE Transactions on Software Engineering**, Vol. SE - 13, No. 12, December 1987, pp. 1278 - 1296.

Beizer, Boris. **Software Testing Techniques**. Van Nostrand Reinhold, New York, 1983.

Blum, B.I. "On Data and Their Analysis." **ACM SIGSOFT SOFTWARE ENGINEERING NOTES**. Vol 14, No. 5, July 1989, pp. 24 - 34.

Cavano, Joseph P. "Toward High Confidence Software." **IEEE Transactions on Software Engineering**. Vol SE-11, No. 12, December 1985.

Deitel, H. M. **Operating Systems**. Addison-Wesley Publishing, Reading, MA, 1990.



DeMillo, Richard A., Lipton, Richard J. and Sayward, Frederick G. "Hints on Test Data Selection: Help for the Practicing Programmer." **Computer**. April 1978, pp. 34 - 41.

Dijkstra, Edsger W. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs." **Communications of the ACM**. Vol 18, No. 8, August 1975, pp. 453 - 457.

"Glossary of Software Engineering Terminology." **ANSI-IEEE Std 729-1983**, Institute of Electrical and Electronics Engineers, 1983.

Gries, David. "An Illustration of Current Ideas on the Derivation of Correctness Proofs and Correct Programs." **IEEE Transactions on Software Engineering**. Vol SE-2, No. 4, December 1976, pp. 238 - 244.

Hetzel, W.C. "An Experimental Analysis of Program Verification Methods." Ph.D. Dissertation, University of North Carolina at Chapel Hill, Chapel Hill, NC, 1976.

Hoare, C. A. R. "An Axiomatic Basis for Computer Programming." **Communications of the ACM**. Vol 12, No. 10, October 1969, pp. 576 - 583.

Kim, K. H. and Welch, Howard G. "Distributed Execution of Recovery Blocks: An Approach for Uniform Treatment of Hardware and Software Faults in Real-Time Applications." **IEEE Transactions on Computers**, Vol 38, No. 5, May 1989, pp. 626 - 636.

King, James C. "Symbolic Execution and Program Testing." **Communications of the ACM**. July 1976, Volume 19, No. 7, pp. 385 - 394.

Lamb, David Alex. **Software Engineering : Planning for Change**. Prentice Hall, Englewood Cliffs, NJ, 1988.

Myers, Glenford J. **The Art Of Software Testing**. John Wiley and Sons, New York, NY, 1979.

Neumann, Peter G. "Letter from the Editor." **ACM SIGSOFT : Software Engineering Notes**. Vol 14, No. 1, January 1989, pp. 5 - 21.

Neumann, Peter G. "RISKS : Cumulative Index of Software Engineering Notes." Vol 14, No. 1, January 1989, pp. 22 - 26.

Reifer, D. J. "The Software Engineering Checklist." **Proceedings, Computers in Aerospace Conference**. 1977, pp. 125 - 129.

Shelly, Gary B. and Cashman, Thomas J. **COMPUTER FUNDAMENTALS for an Information Age**. Anaheim Publishing Company, Brea, CA, 1984.

Shimeall, Timothy. "FALTER - A Fault Annotation Tool." Technical Report NPS52-89-051, Naval Postgraduate School, Monterey, CA, September, 1989.

Shimeall, Timothy. "REACHER - A Reachability Condition Derivation Tool." Technical Report NPS52-89-050, Naval Postgraduate School, Monterey, CA, September, 1989.

Shimeall, Timothy. "An Empirical Comparison of Software Fault Tolerance and Fault Elimination." PhD Dissertation, University of California, Irvine, CA, 1989.

Shimeall, Timothy and Griffin, Rachel. "A Tool Set for the Analysis of Software Failure Regions." Submitted for publication, 1989.

Shimeall, Timothy and Leveson, Nancy. "An Empirical Comparison of Software Fault Tolerance and Fault Elimination." Technical Report NPS52-89-047, Naval Postgraduate School, Monterey, CA, July 1989.

Voas, Jeffrey M. and Morell, Larry J. "Fault Sensitivity Analysis (PIA) Applied to Computer Programs." Technical Report WM-89-4, College of William and Mary, Williamsburg, VA, 10 December 1989.

Voas, Jeffrey M. and Morell, Larry J. "Propagation Characteristics through Versions of One Program." Technical Report WM-89-3, College of William and Mary, Williamsburg, VA, 13 October 1989.

Voas, Jeffrey M. and Morell, Larry J. "Infection and Propagation Analysis : A Fault-Based Approach to Estimating Software Reliability." Technical Report WM-88-2, College of William and Mary, Williamsburg, VA, 14 September 1988.

Wilson, James. "Problem Solving Strategies in A Fault Diagnosis Task." PhD Dissertation, Pennsylvania State University, PA, 1985.

## INITIAL DISTRIBUTION LIST

- |    |   |    |
|----|---|----|
| 1. | Defense Technical Information Center<br>Cameron Station<br>Alexandria, VA 22304-6145  | 2  |
| 2. | Library, Code 0142<br>Naval Postgraduate School<br>Monterey, CA 93943-5002  | 2  |
| 3. | MAJ J. Manning Bolchoz<br>1025 Chambers Lane<br>Mt. Pleasant, SC 29464  | 4  |
| 4. | Timothy Shimeall<br>Naval Postgraduate School<br>Code 52, Department of Computer Science<br>Monterey, CA 93943-5100                 | 10 |
| 5. | LCDR Rachel Griffin<br>Naval Postgraduate School<br>Code 52, Department of Computer Science<br>Monterey, CA 93943-5100              | 1  |
| 6. | Robert B. McGhee<br>Chairman, Department of Computer Science<br>Naval Postgraduate School<br>Monterey, CA 93943-5100                | 1  |
| 7. | CDR Thomas J. Hoskins<br>Curricular Officer, Department of Computer Science<br>Naval Postgraduate School<br>Monterey, CA 93943-5100 | 1  |
| 8. | Paul E. Ammann<br>School of Engineering and Applied Science<br>University of Virginia<br>Charlottesville, VA 22093                  | 1  |

- |     |  |   |
|-----|--|---|
| 9.  | Susan Gerhardt<br>MCC Software Technology Program<br>3500 W. Balcones Drive<br>Austin, TX 78759                          | 1 |
| 10. | Richard Kemmerer<br>Department of Computer Science<br>University of California, Santa Barbara<br>Santa Barbara, CA 93106 | 1 |
| 11. | John Knight<br>Software Productivity Consortium<br>Reston, VA 22091  | 1 |
| 12. | Nancy Leveson<br>ICS Department<br>University of California, Irvine<br>Irvine, CA 92717                                  | 1 |
| 13. | Larry J. Morell<br>Department of Computer Science<br>College of William and Mary<br>Williamsburg, VA 23185               | 1 |
| 14. | Debra Richardson<br>ICS Department<br>University of California, Irvine<br>Irvine, CA 92717                               | 1 |
| 15. | Jeffrey M. Voas<br>Department of Computer Science<br>College of William and Mary<br>Williamsburg, VA 23185               | 1 |
| 16. | Elaine Weyuker<br>Department of Computer Science<br>Courant Institute of Mathematical Science<br>New York, NY 10012      | 1 |